THE UNIVERSITY OF MICHIGAN

Technical Report 22

AN IMPLEMENTATION OF THE QUEUE ANALYZER SYSTEM (QAS) ON THE IBM 360/67

L. S. Randall
I. S. Uppal
G. A. McClain
J. F. Blinn

CONCOMP: Research in Conversational Use of Computers
F. H. Westervelt, Project Director
ORA Project 07449

supported by:

ADVANCED RESEARCH PROJECTS AGENCY
DEPARTMENT OF DEFENSE
WASHINGTON, D.C.

CONTRACT NO. DA-49-083 OSA-3050
ARPA ORDER NO. 716

administered through:

OFFICE OF RESEARCH ADMINISTRATION ANN ARBOR

May 1970

ABSTRACT

This report details and documents QAS, a conversational programming system composed of an aggregation of programs and data structures resident in the IBM 360/67 which accepts graphical descriptions of Markovian queueing networks via data-phone from a remote graphical system resident in a DEC 339, and which returns solutions to these networks to the remote system according to requested specifications.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

I. Introduction

The purpose of this report is to document implementation of the Queue Analyzer System (QAS), the theory and the rationale of which appear in the reports "A System for the Solution of Simple Stochastic Networks" by K. B. Irani and V. L. Wallace [1] and "Network Models for the Conversational Design of Stochastic Service Systems" by V. L. Wallace and K. B. Irani [5].

Briefly, QAS is an aggregation of programs and data structures resident in the IBM 360/67 which accept information concerning the construction of simple stochastic networks sent via data-phone from a remote system called SELMA [2] resident in a DEC 339 and which return solutions to these networks to SELMA in a "conversational" manner.

QAS consists of five sets of routines: supervisory and support routines, generation phase routines, compilation phase routines, result phase routines, and documentation phase routines. The supervisory and support routines include: (1) those routines which interact with SELMA to interpret commands received from the DEC 339 and dispatch them to the proper phase routine, to request information from the DEC 339, etc. and (2) those routines which are used in general by all the various phases to manage free core for the various data structures to perform set manipulations on certain data structure, to provide error checking and debugging facilities, etc. The generation phase routines create the data structures representing the stochastic service networks which are described by commands received from SELMA. The compilation phase routines are divided into two groups. The first of these groups operates upon the data structures created by the generation phase routines and reduces (i.e., compiles) these structures to a form more suitable for use by the second group. The second of these groups, operating on the outcome of the compilation, computes a vector of the steady state probabilities for all the states which the stochastic network may assume. This group also creates a data structure representing the multi-dimensional cartesian product state space for the model, along with information which allows mapping from this state space into a linear index for referencing the steady state probability vector. The result phase routines compute and format requested results for display as graphs or printed tables by the remote system, SELMA. Finally routines in the documentation phase save and retrieve QAS data structures and corresponding SELMA display structures for partially or completely solved networks.

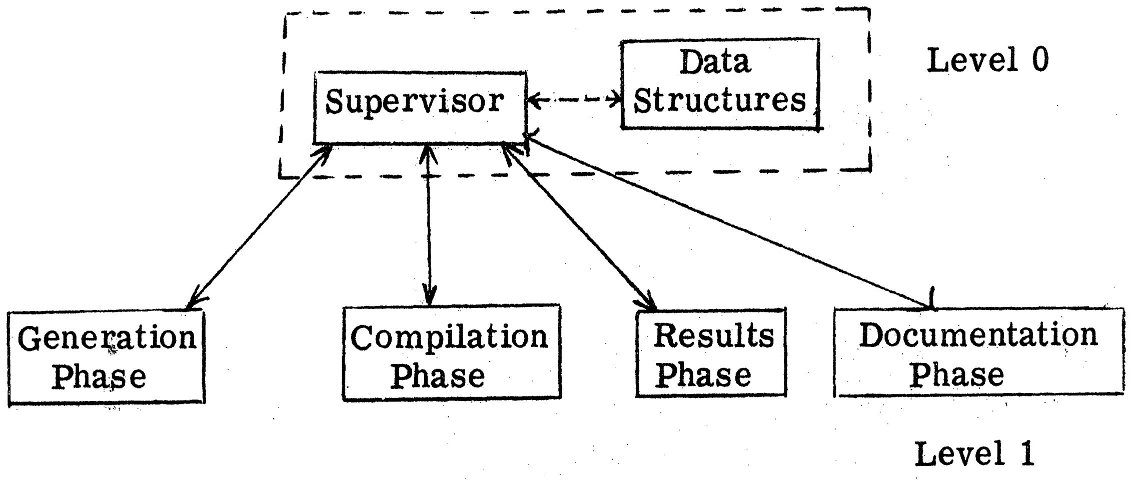
QAS has primarily been written in IBM system/360 Assembler Language (Level-G), except for a few FORTRAN routines for printing results. It has been designed to run under MTS [4], (Michigan Terminal System), which together with UMMPS (University of Michigan Multi-Programming System) provides a time-sharing operating system for an IBM 360/67-2. QAS uses a number of routines provided by the operating systen, and these are listed in Appendix B with their functional descriptions.

In the sections which follow, descriptions of various routuines as well as the operations performed by them are described. Section II describes the supervisory and support routines. The generation phase routines, the compilation phase routines, the results phase routines and the documentation phase routines are described in Sections III, IV, V, and VI, respectively. Finally in Section VII, the results of this work and future modifications are discussed.

II. Supervisory and Support Routines

A. QAS Supervisor

Since under the current accounting scheme used by the MTS [4] system a user gets charged for the virtual (not just the actual) memory used, an overlay technique, using disc files as back up store, was used initially to decrease the virtual memory-elapsed time integral. The QAS supervisor was, of course, the principal root or basic overlay and remained loaded throughout the hook-up time. The rest of QAS was logically segmented into four phases - generation, compilation, results, and documentation. These phases constituted the rest of the overlays and were grouped into the next-to-root level. This structure is pictorially depicted below as an upside-down tree with the supervisor as the root and the control paths as branches.



All storage necessary for the QAS data structures was attached to the supervisor, and was dynamically manipulated (i.e., acquired, released, and expanded) by the supervisor.

The MTS [4] routines LOAD and UNLOAD were used to load and unload level 1 overlays. This process of loading and unloading overlapped however, and seemed to involve overhead significant enough to offset the advantages of keeping the virtual storage requirements to a minimum. Moreover, the delay during loading significantly reduced the response time of QAS. As a result, another version of QAS, which kept all portions of the system resident in the virtual memory, all the time, had to be developed. The rest of this discussion centers around the version without the overlap feature.

Execution of the Queue Analyzer System is invoked by initiating execution of the QAS supervisor through the MTS [4] command

```
$RUN QAS GUSER = ... SERCOM = ... SPRINT = ... 6 = ...
```

where the logical I/O unit references are as follows:

GUSER:	where input records are to be found
SPRING:	where output records are to be placed
SERCOM:	where area (core) dumps are to be placed (if requested).
6:	where requested results are to be printed

In the default case the logical I/O units referenced are, of course, the SINK and the SOURCE (i.e., SELMA).

When the command $RUN QAS is given, the supervisor starts by sending a prompting command to logical I/O unit SPRINT and then attempting to read a logical record consisting of one command from logical I/O unit GUSER. (To avoid line contention problems no command may be sent to QAS until the prompting command has been received.)

When a command is received by the QAS supervisor, it is dispatched to the proper routine for processing. After all operations implied by the command have been carried out, an output buffer is checked to determine whether any output records were produced by these operations. If indeed some records were generated, one of these records is sent to logical I/O unit SPRINT. If no output records were produced, the prompting command is sent again. The supervisor then awaits a record from GUSER. Thus, the supervisor alternately sends and then receives one logical record. Note that this procedure may not be violated. Note one further restriction—no command which results in the generation of output by QAS may be sent to QAS until all records in the output buffer have been transmitted.

During the execution of the current command, if any routine finds an error condition, the QAS supervisor routine PATCHUP is called. PATCHUP responds by sending the following logical record to logical I/O unit SPRINT:

1. A code indicating that "patchup node" has been entered.
2. A "patchup request" - a command which PATCHUP feels will correct the error, sans certain parameters.

The patchup request is generated by the routine which calls PATCHUP and is passed as a parameter. Table 1 lists all the patchup requests and their interpretations.

Another important function served by the supervisor is the management of all the storage required for QAS data structures (hereafter simply referred to as "areas"); all area manipulations are done by the supervisor or through the routines provided by the supervisor. The four major areas of interest are:

1. Network Area

The network area corresponds to the storage necessary for the creation of the data structure representing the stochastic service network during the generation phase. Since the compiler operates destructively on the network area, a copy of the network area must be retained to allow subsequent modification of the model.

2. Working Network Area

The working network area, initially, is simply a copy of the network area, and it contains the structure upon which the compiler operates during the compilation phase. Data structures corresponding to the state space and transition intensity matrix of the matrix of the model are generated in this area. The analysis subroutines also use this area

Table 1

Patchup Requests

Request (Hexadecimal)	Interpretation
01, 03, 80, 60, 00, EN, EPN	Port EPN of Element EN is unconnected.
01, 03, CN, CPN, 00, 00	Port CPN of connection CN is unconnected.
01, 02, NAME, PN	PNth parameter for the element/ connection, whose NAME is specified is either missing or messed up.
04, 00	File specified in documentation command SAVE, does not exist or is not available.
04, 02	File specified in documentation command RETRIVE, does not exist or is not available.

to solve for the vector of steady state probabilities.

3. Results Area

The analysis subroutines in the compilation phase (i.e., those in the second group) create the results area through appropriate calls to the supervisory subroutines. The results area then contains the vector of the steady state probabilities for all the states of the model and a data structure representing the multi-dimensional cartesian product state space for the model. The results area is also used by the result phase routines to compute the results requested by SELMA.

4. Working Probability Area

This is just a scratch vector used by the analysis routines during the iteration cycle.

Note that after compilation, the only pertinent information resides in the results area, and both the working network area and the working probability area are released. Also the results in the results area are constrained to correspond to the network described by the network area. Thus, if there are any subsequent changes in the model, the previous results area is destroyed and lost unless there was an explicit request from SELMA to save the previous model and the results area corresponding to it on some file. By checking the status of the results area (existent or nonexistent) the supervisor can check whether the most up-to-date model, as described by the network area, has already been compiled or not.

The general format for all the traffic between QAS and SELMA is as follows:

$$\boxed{D_b}\boxed{P}\boxed{C}\underbrace{\boxed{\quad}\boxed{\cdots}\boxed{\quad}}_{\text{Parameters}}\boxed{D_e}$$

where

D_b	is a delimiting code of one byte (X'FF') indicating the beginning of the command,
D_e	is a delimiting code of one byte (X'FF') indicating the end of the command
P	is a code of one byte indicating which phase (i.e., network generation phase, compilation phase, etc.) the command applies to,
C	is a code of one byte indicating the actual command within the phase and

the parameters, if any, are those pieces of information which are necessary to the execution of the command.

During both the patchup mode and the normal mode, commands to QAS are dispatched in the following manner by the command dispatcher of the supervisor: The second byte of the command string, P, is examined and the phase to which the command applies is determined according to Table 2. Then depending upon the phase of the command, the following action is taken.

Table 2

QAS Phase Codes

Phase	Command Phase Code Byte, P (Hexadecimal)
Control Phase	00
Network Generation Phase	01
Compilation Phase	02
Result Phase	03
Documentation Phase	04

Control Phase Commands:

The remainder of the command (the third byte C) is scanned and proper action as determined by Table 3 is carried out.

Network Generation Phase Commands:

If the command dispatched previously was from the generation phase, control is transferred to the generation phase interpreter, a routine named GENERATE. Otherwise, the results area is destroyed if it exists, the network generation area is created if it does not exist, and then control is transferred to GENERATE.

Compilation Phase Commands:

Before dispatching a command for this phase, the supervisor always inserts into the command stream a generation phase command implying a network check, and dispatches it. If the network check is successful, the compilation command is dispatched; otherwise patchup mode is entered and the compilation command is lost. After checking the network, the working network area is created and control is transferred to the compilation phase routine COMPSOLV.

Result Phase Commands:

The status of the results area is checked to determine whether the current network has already been compiled. If the results area exists, control is transferred to the result phase interpreter, RESULTS. Otherwise a compilation phase command with the following default parameters is dispatched before transferring control to RESULTS.

Table 3

QAS Control Phase Commands

Command Format: $\text{FF00}\underbrace{\text{XX}}_{\text{C}}\text{FF}$

Command Byte, C (Hexadecimal)	Interpretation
00	Prompting command from SELMA— Send next logical record
01	QAS is to enter patchup mode
02	Call SYSTEM
03	Call ERROR
04	Call MTS
05	Initialize QAS for a new network
06	Clear the output buffer

Commands sent by QAS to SELMA

Command (Hexadecimal)	Interpretation
FF0000FF	Prompting command—send next logical record
FF0001. . .FF	QAS has entered patchup mode
FF0002FF	Logical end of file

Maximum number of iterations = 250.

Convergence criterion (i.e., the maximum absolute difference between corresponding elements of successive iterates that must be met before iterations are halted) = 0.0001.

Documentation phase commands:

Control is simply transferred to the documentation phase interpreter.

A number of routines (actually, entries to the supervisor) may be used by the various phase programs for the management of output buffers and area. These routines, descriptions of which follow below, are DAMMIT, PATCHUP, MOVEIN, TAKEOT, SAVE, ACQUIR, RELESE, STATUS, and QASWRITE.

DAMMIT

DAMMIT is called if any non-recoverable error is found by any routine. Errors of this nature should be found only during the debugging phase of developing the system and, hence, DAMMIT should never be called once the system is operational. Passed to DAMMIT is a single parameter, an error code, the possible values of which appear in Table 4. When called, DAMMIT produces a 17-word hexadecimal dump, the first 16 words of which represent the contents of the general registers at the occurrence of the error and the last word of which represents the error code. At this point QAS enters patchup mode and requests SELMA to initiate a command to call ERROR.

Calling Sequence:[*]

GPR 15:	Address of DAMMIT
GPR 1:	Address of one-word parameter list.

Parameter List:

Word 1:	Location of full-word hexadecimal error code.

----

[*] The listed general-purpose registers are loaded with the information indicated, general-purpose register 13 is loaded with the address of a save area, the routine is entered via a BASR 14, 15, and if a return is to be made, it is made via a BR 14. All general purpose registers may be assumed to be restored upon return unless otherwise indicated.

Table 4

DAMMIT Error Codes

Error Code (Hexadecimal)	Detecting Routine
000000A0	@CROSSI
000000B0	@CIRCDOT
00000100	@GET
00000200	@FREE
00000400	@FNAME
00000500	MOVEIN, TAKEOT
00000600	GENERATE
00000700	DISPATCH
00000800	QAS
00000A00	QAS
00000B00	PATCHUP
00000C00	QASWRITE
00000D00	COMPSOLV
00002000	PUTPRVEC
00002100	SETPRIOR
00004000	DOCUMENT
00004100	DOCUMENT
00005000	RESULTS
00005100	PLOT
00010000	Set manipulation routines
00110000	CREAT
00120000	CONNCT
00130000	ASSPAR
00140000	DISCON
00150000	DSTRY
00160000	ALTER
00170000	Network generation phase routines

Table 4 (Continued)

DAMMIT Error Codes

Error Code (Hexadecimal)	Detecting Routine
00210000	ABSORB
00220000	SPON
00230000	ABSORB
00240000	ABSORB
00250000	SPON
01000000	@ELTYPE
02000000	@TEST
11000000 through 18000000	ANUMCV

PATCHUP

PATCHUP is called to enter patchup mode.

Calling sequence:

GPR 15:	Address of PATCHUP
GPR 1:	Address of two-word parameter list.

Parameter list:

Word 1:	Location of full word containing length of patchup request (command prototype).
Word 2:	Location of patchup request.

MOVEIN

MOVEIN obtains a block of storage for a given area and then copies the indicated area file into the storage obtained.

Calling sequence:

GPR 15:	Address of MOVEIN
GPR 1:	Address of three-word parameter list.

Parameter list:

Word 1:	Integer name of area.
Word 2:	Integer number of logical records contained in area file. (Each logical record is 240 bytes long.)
Word 3:	Location of first character of file or device name from which area file is to be obtained. Name must be terminated by a blank, but has no alignment

requirements.

MOVEIN returns the location of the storage obtained (and hence the location of the given area) in general purpose register 0.

TAKEOT

TAKEOT empties a given file and copies a given area in storage into that file in logical records of 240 bytes. If the file name given is the name of a file that does not exist, the file will be created only if the name is that of a temporary file name. The storage associated with the area is released.

Calling sequence:

GPR 15:	Address of TAKEOT
GPR 1:	Address of two-word parameter list.

Parameter list:

Word 1:	Integer name of area.
Word 2:	Location of first character of file or device name. Name must be terminated by a blank.

SAVE

SAVE performs the same operations and has the same calling sequence as TAKEOT except that the storage associated with the area is not released.

ACQUIR

ACQUIR obtains a block of storage for a given area and inserts a pointer to it in the proper entry of the area table. (This is essentially an initialization operation for the given area.)

Calling sequence:

GPR 15: Address of ACQUIR

GPR 0: Integer name of area.

GPR 1: Size of area in units of 240 bytes.

Upon return from ACQUIR general purpose register 0 contains the location of the storage obtained.

RELESE

RELESE releases the storage associated with a given area. (This is essentially a destroy operation for the given area.)

Calling sequence:

GPR 15: Address of RELESE

GPR 0: Integer name of area.

STATUS

STATUS acquires the location and the size of a given area. It is through this routine that the supervisor transfers addresses of areas to the phase programs.

Calling sequence:

GPR 15: Address of STATUS

GPR 1: Integer name of area.

Upon return from STATUS general purpose register 0 contains the size of the area in units of 240 bytes (if the area exists) and general purpose register 1 contains:

(a) the location of the area if it exists in virtual memory,

(b) hexadecimal 'FFFFFFFF' if the area does not exist.

QASWRITE

QASWRITE inserts an output record in the output buffer. Only the records which are to be interpreted by SELMA (i.e., commands for SELMA) are stuffed in the buffer. Routines which print results on the teletype bypass this routine completely. Also each record must not exceed 120 bytes. As noted earlier, since no command which results in the generation of output by QAS may be sent to QAS until all records in the output buffer, due to previous command, have been transmitted, a non-cyclic buffer, whose size is dynamically increased by 4096 bytes anytime there is no room to insert additional records, is maintained.

Calling sequence:

GPR 15: Address of QASWRITE

GPR 1: Location of record

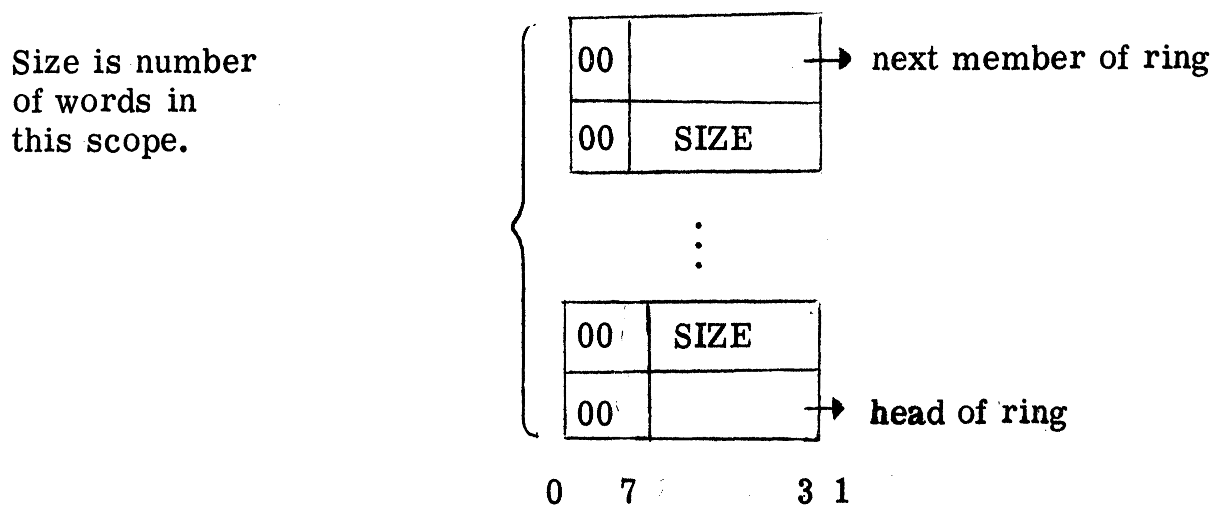
GPR 0: Size of the record in bytes.

B. Free Block Manager

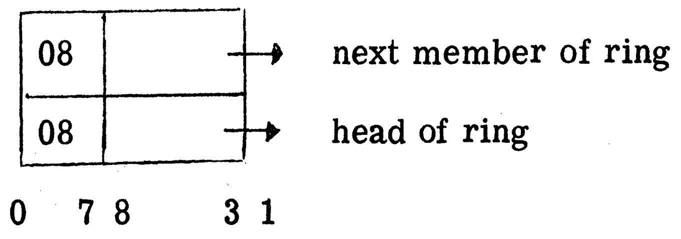
The Free Block Manager is a collection of routines which take care of the bookkeeping involved in using and freeing pieces of storage within the various areas. Besides providing this garbage collection facility, the free block manager also performs such functions as relocating areas and dynamically increasing the size of the areas. It uses sophisticated algorithms to acquire and free pieces of storages to avoid excessive splintering of storage in a given area.

The Free Block Manager keeps track of all free (that is, currently unused) core in a given area by inserting all free blocks of storage within the area into a number of category rings. To this end, an area is a large named vector, the first word of which contains the length of the area in bytes and the next eight words of which form the heads of eight category rings whose members are free blocks in eight ranges of size. The formats used by free blocks in the various category rings are shown below:

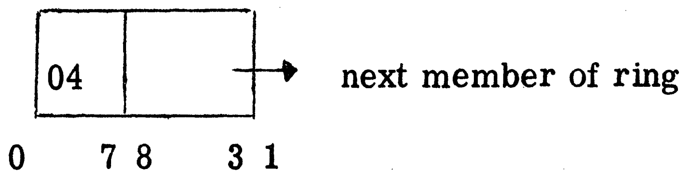
Blocks consisting of three or more words:



Blocks consisting of two words:



Blocks consisting of one word:



Note that all pointers used in the category rings and, for that matter, in all structures created in an area are twenty-four bit displacements relative to the address of the first word, or base, of the area. This convention allows complete relocatability of the areas.

Because of their special formats, blocks of one and two words are segregated from all other blocks and kept in the first category ring. The size ranges for the first seven category rings are specified in words 10 through 16 of the area, respectively. The left half-word of a given word designates the lower limit on the size of blocks in the given ring and the right half-word designates the upper size limit. Note that the size ranges do not overlap. The eighth, or last, category ring contains miscellaneous blocks, i. e., all those blocks whose sizes do not fall within the range of the first seven category rings. Currently, the size ranges of the various category rings are as follows:

Category	Lower Limit	Upper Limit
1	1 word	2 words
2	3 words	5 words
3	6 words	10 words
4	11 words	20 words
5	21 words	40 words
6	41 words	80 words
7	81 words	160 words
8	miscellaneous	

Word 17 of the area contains the size of the most commonly used block. This information is used to avoid unnecessary splinteringof the storage in a given area.

In certain instances it is desirable to acquire a rather large block of storage within an area and then to further subdivide this block as if it were an area itself with the capability of freeing the large block at a later time without having to free each subblock within it first. Hence, this facility has been incorporated into the Free Block Manager. The area itself is referred to as the <u>major</u> area and the block within it to be treated as an area is called the <u>minor</u> area. When the minor area is acquired, the category information of the major area is saved and replaced in the area by new category information concerning the minor area. Upon releasing the minor area, the major area information is restored.

The Free Block Manager has the following entries: SETUP, SETMINOR, GMAJOR, GMINOR, @GET, @FREE, and ERFREE. Descriptions of these entries follow below.

<u>SETUP</u>

SETUP is used to initialize the category rings of a major area. All category rings are made empty except for the miscellaneous category ring which contains one block composed of all the storage in the area not used by the category ring heads and their related information.

Calling sequence:

GPR 15:	Address of SETUP
GPR 1:	Address of four-word parameter list.

Parameter list:

Word 1:	Location of major area.
Word 2:	Location of full word containing size of area in words.
Word 3:	If nonzero, size in words of most commonly used block.
Word 4:	If nonzero, size in 240-byte units of additional storage to be acquired if current storage is used up.

If word 4 of the parameter list is zero or negative and the routine @GET is unable to find requested space, the QAS error routine DAMMIT is called with error code X'00000100'.

SETMINOR

SETMINOR is used to establish a particular block as a minor area. The major area category ring heads are saved and then replaced by new heads, all of which have empty rings except for the miscellaneous category which contains the minor area. Normal operation of the Free Block Manager now segments the minor area. The original category heads may be restored by a call to GMAJOR. Note that only one minor area may be set up at a time.

Calling sequence:

GPR 15:	Address of SETMINOR
GPR 0:	Size of minor area in words.
GPR 1:	Pointer to minor area with respect to base of major area.
GPR 12:	Location (base) of major area.

GMAJOR

GMAJOR is used to restore the category ring heads of a major area and at the same time to save the current category ring heads which correspond to some minor area contained within the major area.

Calling sequence:

GPR 15:	Address of GMAJOR
GPR 12:	Location (base) of major area.

GMINOR

GMINOR is used to return to the category ring heads of an existing minor area. (A typical example of its use is where a minor area if set up initially by a call to SETMINOR, the major area category ring heads are restored by a call to GMAJOR, and the user wants to return to using the minor area.)

Calling sequence:

GPR 15: Address of GMINOR

GPR 12: Location (base) of major area.

@GET

The purpose of @GET is to find a free block of storage of a specified size. Initially @GET looks for a block of the exact size desired. If it is unsuccessful in its search, @GET looks for a block such that the remainder after splitting off the desired block has a size equal to that of the most commonly used block. If it is still unsuccessful in its search, @GET looks for a block of any size larger than the one specified and frees the remainder after the split. If a block of size greater than or equal to the one requested is not available, the area is moved into a larger area. Additional storage to be acquired is determined by word 4 of the parameter list for SETUP. If no additional storage can be acquired, DAMMIT is called with error code X'00000100'. If the storage in a minor area is exhausted, recovery routine EXPMINOR is called.

Calling sequence:

GPR 15: Address of @GET

GPR 0: Size in words of block to be acquired.

GPR 12: Location (base) of major area from which block is to be acquired (even if block is to be acquired from minor area within it)

Upon return GPR 1 contains a pointer (relative to the base of the major area) to the block acquired and GPR 12 contains the base of the major area. (Note that the value of GPR 12 may change from entry to return if the area is expanded.)

@FREE

@FREE is used to return an unused block of storage to the appropriate category ring. If either or both of the blocks adjacent (in storage) to the freed block is free, the blocks are concatenated. This tends to reduce any possible splintering of free storage. Calling sequence:

GPR 15:	Address of @FREE.
GPR 0:	Size in words of block to be freed.
GPR 1:	Pointer to block to be freed.
GPR 12:	Base of major area in which block resides.

ERFREE

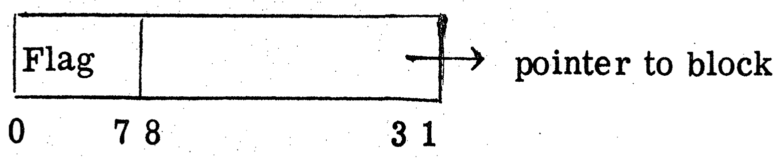
ERFREE is used as a switch to determine what action should be taken in the event that a block which does not exist in a given area is freed in that area. Normally, DAMMIT is called by @FREE with error code X'00000200' if this condition arises, but if ERFREE has been previously called, this error condition results in no operation, i. e., nothing is freed and return is made to the calling program by @FREE.

Calling sequence:

GPR 15: Address of ERFREE.

C. Minor Area Manager

As noted earlier, provisions have been incorporated into the free block manager to allow the acquisition of a rather large block of storage within an area, and then to allow the acquisition and release of sub-blocks within this block. The primary motive for including this feature is to prevent excessive calls to @FREE. Thus, a rather large block of storage can be obtained within an area, the facilities of the free block manager can be used to manage it, and when it is no longer needed, it can be freed by just one call to @FREE. This facility is used by the routine ABSORB, in the compiler, to set up rings of spontaneous events. (This is explained later in the section on the compiler.) Since the storage requirement for a minor area (as used by ABSORB) is a function of the stochastic network to be solved, rather than use an ad-hoc measure for size of the minor area, the minor area may be expanded dynamically. Note that since a minor area is contained within a major area (all pointers are relative to the base of major area) and since it is used only for ring structures representing spontaneous events, it need not consist of a single contiguous block. Furthermore, the minor area is used each time a new connection is absorbed, with the result that the storage requirement of the minor area normally keeps on increasing with the absorption of connections. Thus, one block of storage is acquired to be set up as the minor area initially, and then more blocks of the same size are acquired and attached to the minor area, as needed. After the absorption of one connection (the structure in the minor area is no longer useful), the blocks forming the minor area are not released but rather are reused for the absorption of the next connection. At the end of compilation, all of the blocks in the minor area are released at once. A table of all blocks in the minor area is kept to facilitate this. Each entry in the table contains a 24-bit pointer to one of the blocks and an 8-bit flag indicating the status of the block, as shown below.



Flag	Meaning
X'00'	Corresponding block was obtained previously, but now contains redundant information and can be reused.
X'FF'	Block in use, don't touch it.

A zero entry (both flag and pointer) in the table indicates that the block corresponding to this entry was never acquired. Since all blocks are assumed to have the same size, their sizes are not inserted in the table. All of the above functions are carried out by the routine EXPMINOR.

EXPMINOR

EXPMINOR is used both to set up the first block of a minor area for the absorption of a new connection, and as a recovery routine for @GET to add additional blocks to the minor area when its current storage is exhausted. The content of GPR 0 at the time of a call to EXPMINOR indicates whether the call is a recovery call from @GET or not. The action taken by EXPMINOR is as follows:

For a call to set up the first block in a minor area:

All flags in the block table for the minor area are set to zero. This implies that all the blocks acquired so far (if any) can be reused. Then the table is scanned to determine whether any block has been acquired. If none has yet been acquired, a block is obtained from the major area, SETMINOR is called to set it up as a minor area, and an entry indicating that this has been done is created in the table. If a block which can be reused is available, it is so used and the flag in the corresponding entry in the table is changed to X'FF'. Finally a normal return is made to the calling program.

For a call to expand a minor area:

If @GET is unable to acquire requested storage in a minor area, it calls EXPMINOR in the following (unconventional) manner: @GET restores all general purpose registers and uses a simple branch instruction, instead of branch and link, so that when EXPMINOR gets control, GPR 14 still contains the original return address (for the call to @GET). EXPMINOR then adds another block to the minor area and returns to the start of @GET, again via a simple branch instruction. Using simple branch instructions makes this procedure completely transparent, and @GET proceeds to acquire storage as if the previous blocks were never exhausted. The entries in the block table for the minor area are, of course, updated accordingly. Note that a maximum of ten blocks can be used to set up the minor area. (This restriction is imposed by the size of the block table.)

Calling sequence:

GPR 15:	Address of EXPMINOR
GPR 0:	1) contains 0 if the first block in a minor area is to be set up.
	2) contains 1 if the minor area is to be expanded.
GPR 12:	Base address of the major area.

<u>RESETMIN</u>

RESETMIN clears both the flags and the pointers in the block table and is called at the end of each compilation. It also prints a comment indicating the total amount of storage used in the minor area during the last compilation.

Calling sequence:

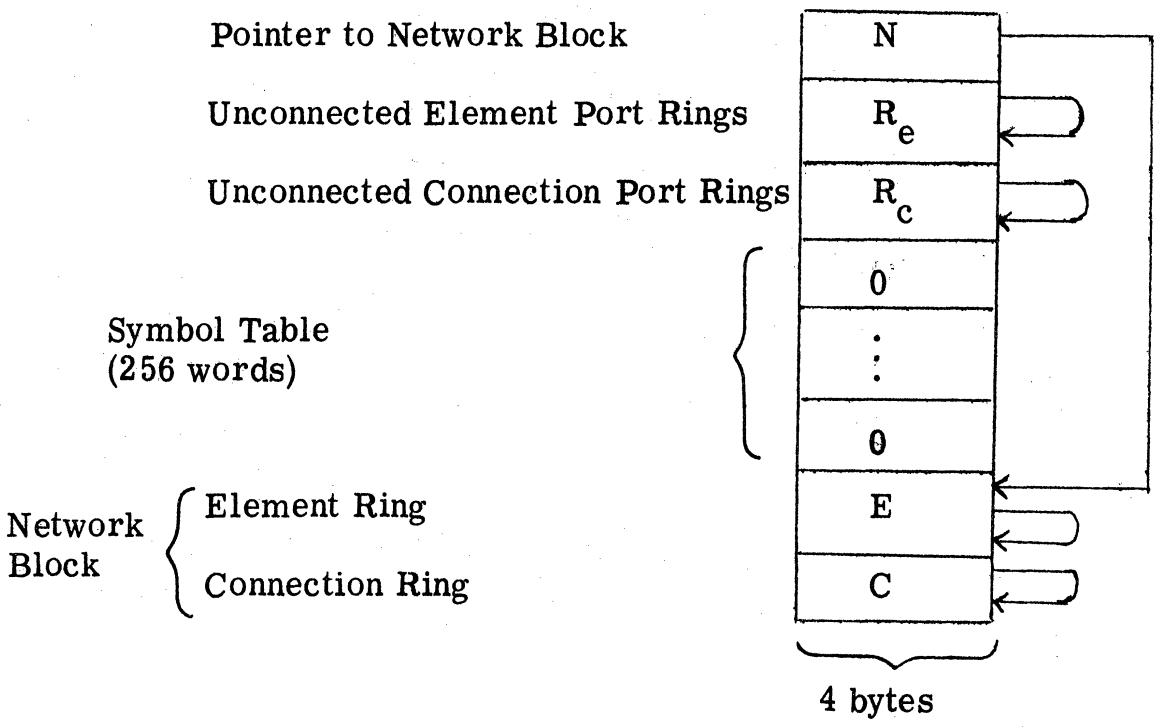
GPR 15:	Address of RESETMIN
GPR 14:	Return address
GPR 12:	Base address of the major area.

D. Symbol Table Service Routines

The symbol table service routines are a set of five routines which are used to set up, to manipulate, and to interrogate a network symbol table resident in the network area. Each symbol, i. e., each element and each connection, in the network diagram is assigned a unique name by SELMA. Element names are assigned from X'01' to X'7F', inclusive, and connection names are assigned from X'81' to X'FF', inclusive. (Note that element name X'00' and connection name X'80' are illegal.) In order to permit rapid, easy access to that part of the network structure which corresponds to a given symbol, a symbol table containing a pointer entry for each symbol is created. (The exact significance of where this pointer points will be considered later.) Physically, the symbol table consists of 256 consecutive words, each of which is accessed by using the symbol name as an index (note that two of these words will never be accessed) and each of which contain a 24-bit pointer relative to the base of the network area. A pointer of zero indicates that no symbol has been assigned the name corresponding to the location of the pointer. The five routines which comprise this group are: @SYMTAB, @FINDPTR, @ENTER, @DELETE, and @FNAME.

@SYMTAB

@SYMTAB is used to initialize the network symbol table and the network structure itself. @SYMTAB must be the first routine called after the call to SETUP (in the Free Block Manager) for the network area. @SYMTAB acquires a block of 261 words displaced 80 bytes from the base of the network area to be used for the symbol table and the initial network structure, which it then sets up as shown below:



The significance of the first three words and the last two words of this structure will be considered later. All of the entries in the symbol table are set to zero since initially, of course, there are no symbols in the network structure.

Calling sequence:

GPR 15: Address of @SYMTAB

GPR 12: Base of network area.

<u>@FINDPTR</u>

Given a symbol name (as defined above) @FINDPTR fetches its associated pointer from the symbol table. (The calling program must check the returned value to insure that it is nonzero.)

Calling sequence:

GPR 15: Address of @FINDPTR.

GPR 0: Name of symbol.

GPR 12: Base of network area.

Upon return GPR 1 contains the pointer associated with the symbol name.

<u>@ENTER</u>

@ENTER is used to enter in the proper slot of the symbol table the pointer associated with a given symbol name.

Calling sequence:

GPR 15: Address of @ENTER

GPR 0: Name of symbol.

GPR 1: Pointer associated with symbol name.

GPR 12: Base of network area.

@DELETE

@DELETE is used to remove (zero out) from the symbol table the pointer associated with a given symbol name.

Calling sequence:

GPR 15: Address of @DELETE

GPR 0: Name of symbol

GPR 12: Base of network area.

@FNAME

@FNAME searches the symbol table for the pointer given and returns the corresponding symbol name. If no such entry exists in the symbol table, DAMMIT is called with error code X'00000400'.

Calling sequence:

GPR 15: Address of @FNAME.

GPR 1: Pointer

GPR 12: Base of network area

Upon return GPR 0 contains the symbol name associated with the given pointer.

E. SEL Six-Bit Coded Numerical Constant Assembler

A routine, called ANUMCV, is available for use in assembling SEL six-bit coded numerical constants. The SEL (Systems Engineering Laboratory) six-bit code, which is described in Table 5, is used for virtually all communications between SELMA and QAS.

ANUMCV is supplied with a string of SEL six-bit coded characteristics (packed one character per byte with leading zeros) which represents some numerical constant of either integer or floating point mode, and with the number of characters in the string. ANUMCV then assembles this constant, returning its value and its mode.

Valid input formats to this assembler are:

For integer constants -

[±][xx...xx]

For floating point constants -

[±][xx...xx][.][xx...xx]E[±][yy]

Each field within brackets is optional under the following restrictions:

(1) The string must contain at least one, but no more than nine, significant digits (the x's), where leading zeros are treated as significant.

(2) A floating point number must contain one or more of the following ".", "E", or the sign preceding the exponent.

(3) If a number is indicated to be floating point by the presence of an "E" or an exponent sign, then the number

Table 5

Systems Engineering Laboratory Six-Bit Character Code

		Second Digit (Octal)							
		0	1	2	3	4	5	6	7
First Digit (Octal)	0	0	1	2	3	4	5	6	7
	1	8	9	A	B	C	D	E	F
	2	G	H	I	J	K	L	M	N
	3	O	P	Q	R	S	T	U	V
	4	W	X	Y	Z	*	/	+	-
	5	(	)	[	]	<	=	>	↑
	6	←	.	,	:	;	?	!	'
	7	"	$	#	&	cr	lf	sp	null

		Second Digit (Hexadecimal)														
		0	1	2	3	4	5	6	7	8	9	A	B	C	D	E
First Digit (Hexadecimal)	0	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E
	1	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U
	2	W	X	Y	Z	*	/	+	-	(	)	[	]	<	=	>
	3	←	.	,	:	;	?	!	'	"	$	#	&	cr	lf	sp

must have at least one but no more than two exponent digits (the y's).

The following violations of these restrictions result in DAMMIT being called with the indicated hexadecimal error codes:

Error Code	Condition
11000000	Nothing follows initial sign.
12000000	Too many significant digits (more than nine).
13000000	More than one "."
14000000	Nothing follows "E" (if present).
15000000	Nothing follows exponent sign (if present).
16000000	Illegal character (Legal characters include digits, "+", "-", ".", and "E", but these must fall in the proper fields. Note that blanks are not legal.)
17000000	Too many exponent digits (more than two).
18000000	No significant digits.

Calling sequence:

GPR 15: Address of ANUMCV

GPR 0: Number of characters in string.

GPR 1: Address of first character in string.

Upon return GPR 0 contains the assembled constant and GPR 1 contains its mode, where X'00000000' implies integer mode and X'00000004' implies floating point mode.

F. Subroutine for Set Manipulation

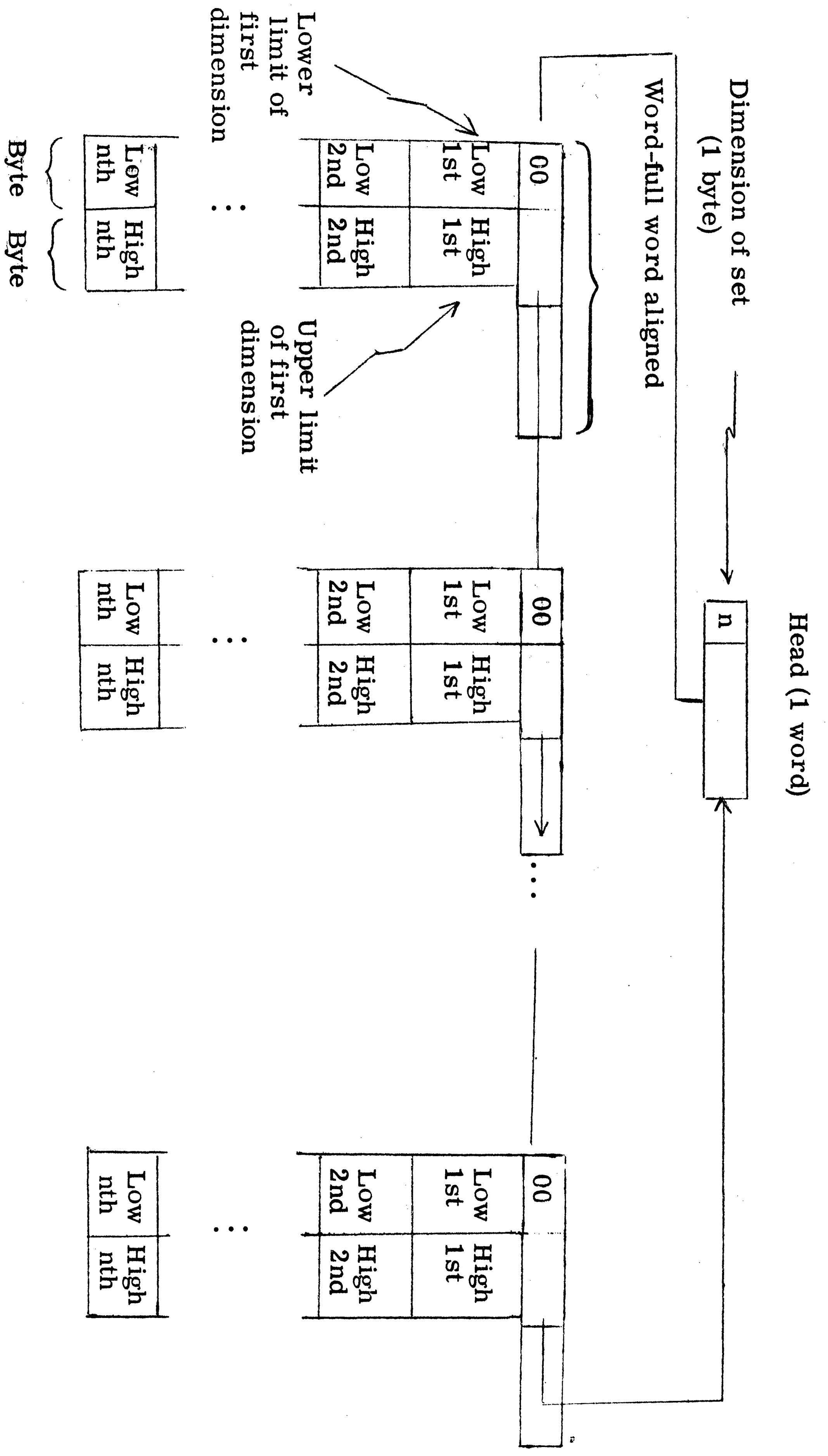
A subroutine is available which has entries to perform certain set operations: namely, cartesian product, intersection, difference, a special operation which yields both intersection and difference, a second special operation for subtracting a vector from a set, and a third special operation for adding a vector to a set. The entry points which correspond to these operations are PRDT, INTR, DIFF, DIFINR, SUBVECT, and ADDVECT, respectively.

The sets (or vector, where applicable) used as operands for these operations are left unchanged and storage for the resultant structure is obtained through @GET.

A set, as used in these operations, is represented in storage as a ring. Each element of the ring specifies an n-dimensional "rectangular" subset of the points in the set. This is done by specifying a low and a high limit for each dimension, effectively specifying the range of values assumed by each dimension in the subset. Thus, in general, a set would appear as shown in the diagram on the next page.

Each ring element consists of consecutive bytes of storage assigned left-to-right, top-to-bottom in the above-mentioned diagram. Furthermore, the following limitations are placed on values associated with a set:

(1) The dimension of the set and each limit value are stored as one-byte binary numbers. Hence,



Dimension of set
(1 byte)
Head (1 word)
n
Word-full word aligned
00
Lower limit of first dimension
Upper limit of first dimension
Low 1st
High 1st
Low 2nd
High 2nd
Low nth
High nth
Byte
Byte

1 dimension of the set $\leq$ 255

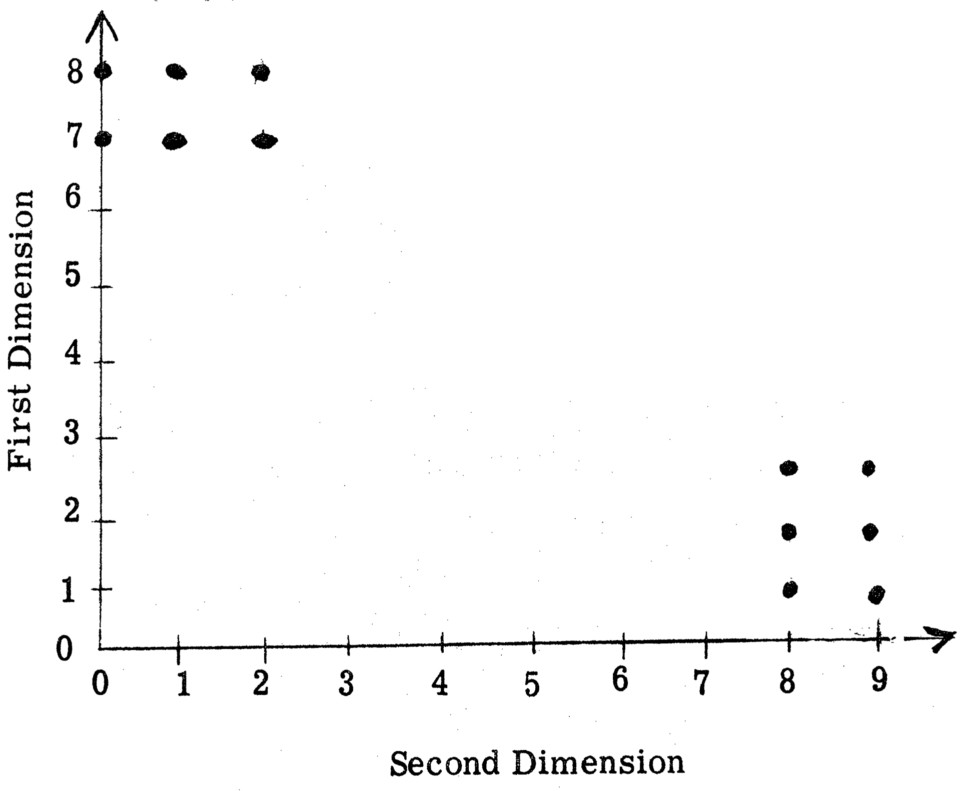
0 $\leq$ limit value $\leq$ 255.

(2) The ring head is one full word.

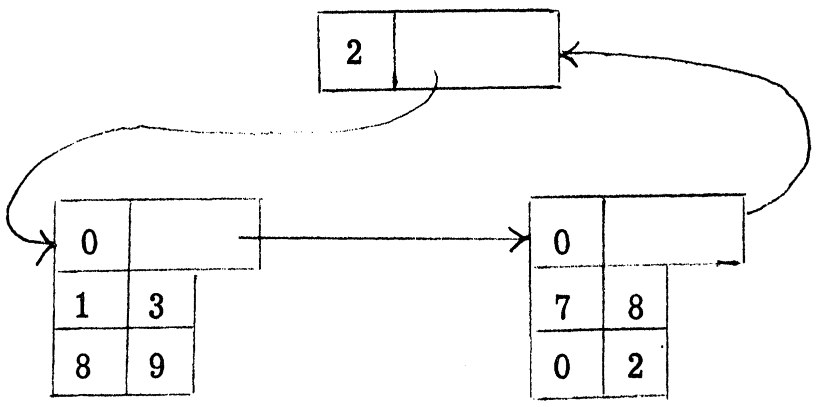
(3) An element of the ring starts on a full-word boundary and requires (2*n + 4) bytes of storage.

All pointers used in the rings (as in all other QAS data structures) are twenty-four bit displacements relative to the area base.

The following is an example of how the two-dimensional set $\{(1,8), (1,9), (2,8), (2,9), (3,8), (3,9), (7,0), (7,1), (7,2), (8,0), (8,1), (8,2)\}$ which may be drawn schematically as



would be represented in ring structure form:



Descriptions of each of the operations available follow below.

PRDT

PRDT generates the set C representing the cartesian product of two sets A and B, A × B. The dimension of the result C is equal to the sum of the dimensions of A and B and the number of elements in the result set ring is equal to the product of the numbers of elements in the set rings for A and B.

Calling sequence:

GPR 15:	Address of PRDT
GPR 1:	Address of three-word parameter list.
GPR 12:	Base of area.

Parameter list:

Word 1:	Address of pointer to head of A ring
Word 2:	Address of pointer to head of B ring

Word 3: Address of word into which pointer to head of C ring is to be placed.

If the result set is empty, an integer 1 is returned in GPR 0; otherwise a zero is returned in GPR 0.

INTR

INTR produces the set C representing the intersection of two sets A and B, $A \cap B$. The dimension of the result set is the same as the dimensions of A and B (which must be the same), and the number of elements in the result set ring is not greater than the product of the numbers of elements in A and B.

Calling sequence:

GPR 15:	Address of INTR
GPR 1:	Address of three word parameter list.
GPR 12:	Base of area.

Parameter list:

Word 1:	Address of pointer to head of A ring.
Word 2:	Address of pointer to head of B ring.
Word 3:	Address of word into which pointer to head of C ring is to be placed.

If the result set is empty, an integer 1 is returned in GPR 0; otherwise a zero is returned.

DIFINT

DIFINT produces two sets, C and D, as the result of operating upon two operand sets A and B. Result set C represents the difference between A and B, A - B, and result set D represents the intersection of sets A and B, $A \cap B$. Both result sets have the same dimension as that of A and B.

Calling sequence:

GPR 15:	Address of DIFINT
GPR 1:	Address of four-word parameter list.
GPR 12:	Base of area.

Parameter list:

Word 1:	Address of pointer to head of A ring.
Word 2:	Address of pointer to head of B ring.
Word 3:	Address of word into which pointer to head of C ring is to be placed.
Word 4:	Address of word into which pointer to head of D ring is to be placed.

A code returned in GPR 0 has the following interpretations:

Return Code	Interpretation
0	C not empty and D not empty
1	C empty and D not empty
2	C not empty and D empty
	C empty and D empty

For the above described routines a number of error conditions are checked:

Dimension of set A equals zero.

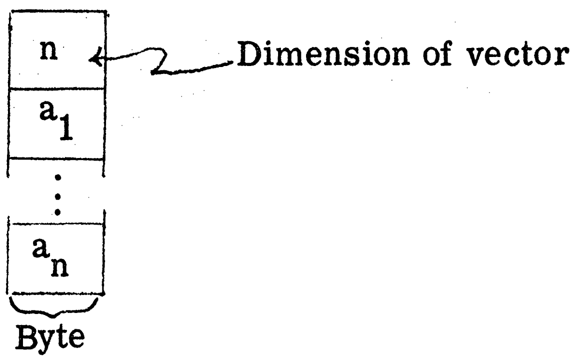
Dimension of set B equals zero.

Sum of dimensions of sets A and B greater than 255 (for PRDT only).

Should one of these conditions occur the routine calls the QAS error routine DAMMIT with error code X'00010000'.

SUBVECT

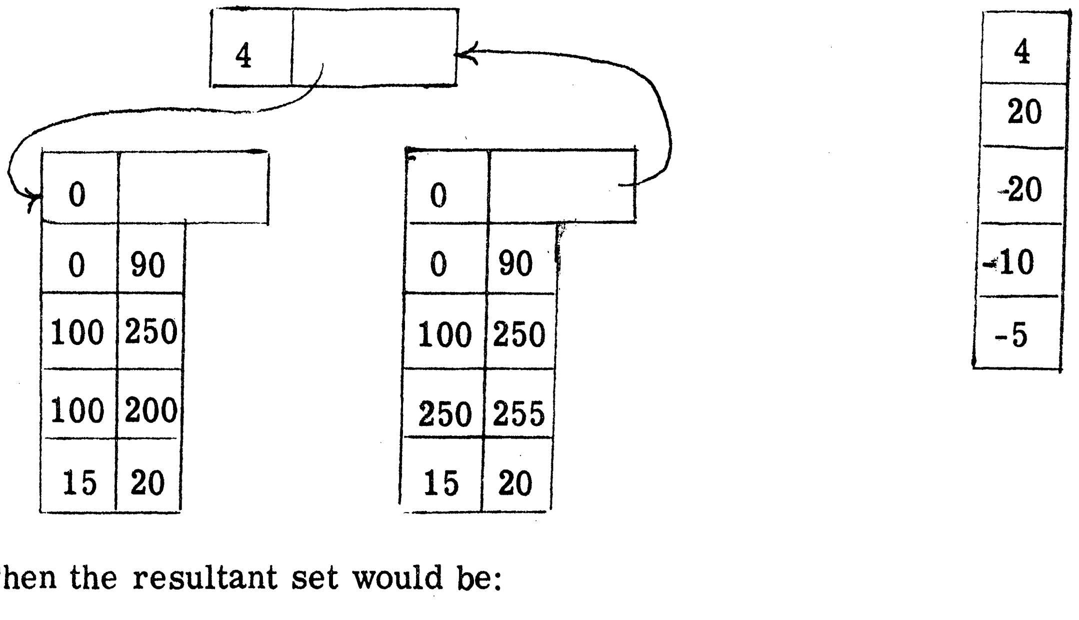
SUBVECT operates upon a vector and a set to produce a new set in which the vector has been "subtracted" from the set. The set assumes the form of sets described above and the vector assumes the following form:



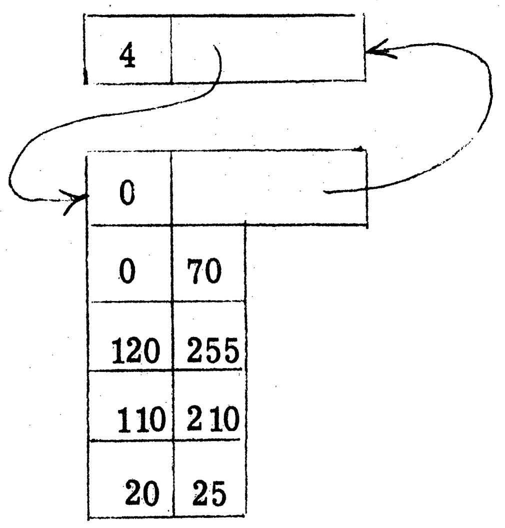
occupying n+1 consecutive bytes of storage. Each dimension of the vector is a one-byte binary number in two's complement notation, thus having a range from -128 to +127.

The "subtraction" operation involves subtracting the vector from both the lower and upper limit "vectors" of each element in the set ring. Recalling that the permissible range of values for the lower and upper limits of the set dimensions is from 0 to +255, it is possible for the subtraction to produce a dimension limit value outside the allowable range of values. In this case the following action is taken: If the resulting lower limit is less than zero, then it is replaced by zero. If the resulting upper limit is greater than 255, then it is replaced by 255. But if the lower limit exceeds 255 or if the upper limit is less than zero, then the entire rectangle (ring element) is deleted from the result set.

For example, suppose that the following set and vector are the operands for this operation:



Then the resultant set would be:



Calling sequence:

GPR 15: Address of SUBVECT.

GPR 1: Pointer to head of set.

GPR 2: Pointer to vector.

GPR 12: Base of Area.

If the result set is empty, an integer 1 is returned in GPR 0; otherwise a zero is returned. Also, on return GPR 1 contains a pointer to the head of the resultant set.

If either the dimension of the set equals zero or the dimensions of the set and the vector are not equal, the QAS error routine DAMMIT is called with error code X'000 10000'.

ADDVECT

ADDVECT operates upon a vector and a set to produce a new set in which the vector has been "added" to the set. The data structures used and the general operation of ADDVECT are the same as those for SUBVECT, the only difference being that ADDVECT involves addition whereas SUBVECT involves subtraction.

Calling sequence:

GPR 15:	Address of ADDVECT
GPR 1:	Pointer to head of set.
GPR 2:	Pointer to vector.
GPR 12:	Base of area.

The return and error detection for ADDVECT are the same as those for SUBVECT.

III. Generation Phase Routines

A. Generation Phase Interpreter

The generation phase interpreter is a routine named GENERATE which is called by the QAS supervisor whenever a generation phase command is received, that is, whenever the first byte of the command (excluding the delimiting bytes) is X'01'. GENERATE checks to insure that the network area (which was supplied by the supervisor) indeed exists in the virtual memory and loads GPR 12 with the base of the network area. If required, GENERATE then calls SETUP and @SYMTAB to initialize the network area. Finally, it examines the second byte of the command and calls the appropriate generation phase routine, according to Table 6.

Calling sequence:

GPR 15:	Address of GENERATE
GPR 1:	Address of one-word parameter list

Parameter list:

Word 1:	Location of first byte of command (including initial delimiter byte).

If the area supplied by the supervisor is not a valid network area or if the second byte of the command is not a legal command code, the QAS error routine DAMMIT is called with error code X'00000600'.

Table 6

Generation Phase Routines

Second Byte of Command (Hexadecimal)	Routine Called	
00	CREAT	command-specific routines
01	DSTRY	command-specific routines
02	ASSPAR	command-specific routines
03	CONNECT	command-specific routines
04	DISCON	command-specific routines
05	ALTER	command-specific routines
06	CKNETWK	
07	HDUMP	

(The exact format of these commands is discussed later on.)

B. Generation Phase Command-Specific Routines

The generation phase command-specific routines are those routines which are used to create the data structure within the network area which represents the network being constructed and displaying by SELMA. This data structure contains all the relational and parametric information necessary for the eventual solution of the problem at hand.

The generation phase command-specific routines, a brief description of which is given in Table 7, have been written as a single multiple-entry routine, the entries of which are (in the order listed in Table 7): CREAT, DSTRY, ASSPAR, CONNCT, DISCON, and ALTER.

A large number of possible error conditions are checked in these routines and in any case where an error is detected, control is transferred to the QAS error routine DAMMIT with the error code indicating which routine detected the error as shown in Table 8. All errors are treated as fatal although many appear at first to be recoverable errors. The rationalization for this is that all such conditions are to be checked first by SELMA so that any such error encountered in QAS may be considered a system error. DAMMIT is called before any change has been made in the network structure.

In addition to DAMMIT these routines make external references to @GET and @FREE when storage is to be obtained or released, and to ANUMCV to convert parameter values from their SEL six-bit coded representations to the IBM 360 arithmetic format.

Table 7

Generation Phase Command-Specific Routine Descriptions

Routine	Input Parameters	Functions
Create Element or Connection	Element or connection name, type, generation parameter values.	Creates element or connection block and associated structure (not including type structure) and joins to network block. Places name in symbol table.
Destroy Element or Connection.	Element or connection area.	Inverse of "Create Element or Connection."
Assign Parameter Values	Element or connection name, parameter number, parameter value.	Inserts parameter value into proper slot of parameter list block (ρ).
Connect	Connection name, connection port number, element name, element port number.	Joins designated port of connection indicated to designated port of element indicated.
Disconnect	Element name, port number.	Replaces disconnected element port and corresponding disconnected connection port on respective unconnected port rings (R_e and R_c).
Alter Generation Parameter	Element or connection name, generation parameter number generation parameter value.	Changes value of generation parameter and regenerates associated structure, preserving existing values and joinings.

Table 8
Generation Phase Error Codes

Error Code (Hexadecimal)	Routine Detecting Error
00110000	CREAT
00120000	CONNECT
00130000	ASSPAR
00140000	DISCON
00150000	DSTRY
00160000	ALTER
00170000	Code common to more than one routine.
00000600	GENERATE

In all of these routines considerable thought was given to making the definition of new element or connection types relatively easy. In general, adding a new type requires inserting a short segment of code and reassembling the routine. This should, however, be straightforward and require a minimum of understanding of the existing code. The routines which will require added code are: CREAT, ASSPAR, and ALTER.

Below follow descriptions of each of the generation phase command-specific routines.

CREAT

CREAT is used to create an element or connection within the network structure. It sets up all necessary storage to represent the particular type of element or connection, places a pointer to the created element or connection block in the symbol table, places the ports of the symbol on the proper ring of unconnected ports , initializes the parameter list block if necessary (described later in this section), and places the element or connection block on the ring of elements or connections, respectively.

Calling sequence:

GPR 15:	Address of CREAT.
GPR 1:	Address of that portion of command string which contains the element or connection name, type number, and any generation parameters.

GPR 12: Base of network area.

Error detected:

Element or connection name already assigned.

Type number undefined.

Element name is X'00' or connection name is X'80'.

DSTRY

DSTRY is used to destroy an element or connection contained within the network structure. It removes the element or connection block from the ring of elements or connections, respectively, removes the ports from the proper ring of unconnected ports, and frees all storage used to represent the element or connection.

Calling sequence:

GPR 15: Address of DSTRY

GPR 1: Address of byte of command string which contains element or connection name.

GPR 12: Base of network area.

Errors detected:

No element or connection block corresponds to name given.

Not all ports disconnected.

Type structure still exists.

ASSPAR

ASSPAR is used to assign a value to the indicated parameter associated with an element or connection. It calls upon ANUMCV to convert the SEL-six-bit coded parameter value contained in the command string to an arithmetic format and then inserts this value into the proper slot of the parameter list.

Calling sequence:

GPR 15: Address of ASSPAR

GPR 1: Address of that portion of command string which contains element or connection name, parameter number, and SEL six-bit coded string representing parameter value.

GPR 12: Base of network area.

Errors detected:

No element or connection block corresponds to name given.

Illegal parameter number.

<u>CONNCT</u>

CONNCT is used to connect an element port to a connection port. The ports are connected after removing them from their respective rings of unconnected ports. The storage used for the connection port is then released.

Calling sequence:

GPR 15: Address of CONNCT

GPR 1: Address of that portion of command string which contains connection name, connection port number, element name, and element port number.

GPR 12: Base of network area.

Errors detected:

No element block corresponds to name given.

No connection block corresponds to name given.

Illegal port number.

Ports are not both disconnected.

DISCON

DISCON is used to disconnect a particular port of a given element from the connection port to which it is connected. A new connection port block is created and the two ports are placed on their respective rings of unconnected ports.

Calling sequence:

GPR 15: Address of DISCON

GPR 1: Address of that portion of command string which contains element name and element port number.

GPR 12: Base of network area.

Errors detected:

No element block corresponds to name given.

Illegal port number

Port already disconnected.

ALTER

ALTER is used to modify a generation parameter for a given element or connection. It makes the changes in the data structures required by the change in the generation parameter. Specifically, this involves altering the length of the parameter list and/or changing the number of ports for the indicated element or connection.

Calling sequence:

GPR 15: Address of ALTER

GPR 1: Address of that portion of command string which contains element or connection name, generation parameter number, and new generation parameter value.

GPR 12: Base of network area.

Errors detected:

No element or connection block corresponds to name given.

No generation parameter associated with this element or connection.

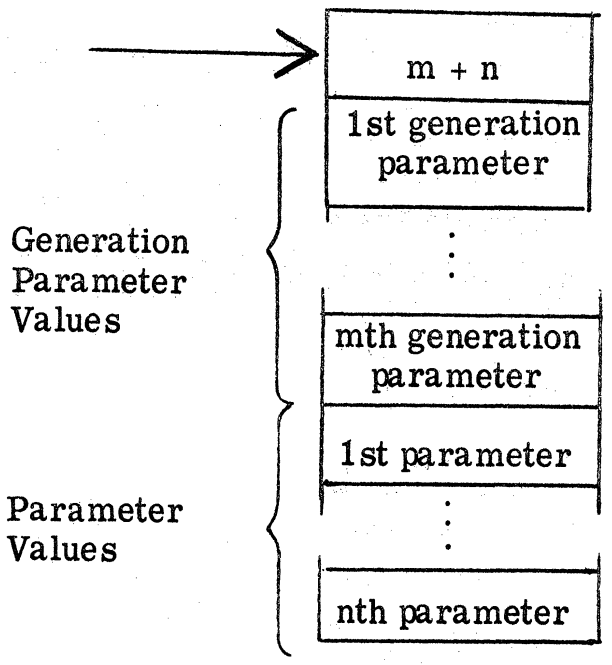
Ports to be deleted (if any) are not disconnected.

Illegal generation parameter number given.

Illegal generation parameter value.

Details of the Generalized Element or Connection Parameter List Block

The parameter list block is a block of $m + n + 1$ words, where m is the number of generation parameters associated with the element or connection and n is the number of parameters. The format of the block is the following:



When the structure representing an element or connection is created (including the parameter list block), the values of the generation parameters are inserted into the parameter list block. Currently only the random branch and priority branch type connections have generation parameters associated with them. In both cases, the generation parameter indicates the number of output ports, as well as the number of parameters. The parameter associated with each output port of a random branch indicates the probability of taking that branch, and in case of priority branch it indicates the priority associated with that branch. Note that the priorities are indicated by the parameters and not by order in which the connection is created. This facilitates changing the priorities in the model.

Generation Phase Command Formats

The generation phase commands, as interpreted by the generation phase interpreter, have the following formats, where each division within the command corresponds to one byte and all numbers contained therein are hexadecimal:

Create Element or Connection

FF	01	00	Name	Type	GPV	...	GPV	FF

Generation Parameters (if any)

Destroy Element or Connection

FF	01	01	Name	FF

Assign Parameter Value

FF	01	02	Name	PN		...		FF

Parameter Value
(In SEL six-bit code,
one character per byte)

Connect

FF	01	03	CN	CPN	EN	EPN	FF

Disconnect

FF	01	04	EN	EPN	FF

Alter Generation Parameter

FF	01	05	Name	GPN	GPV	FF

The abbreviations used within these commands have the following meanings:

Name	-	Element or connection name
Type	-	Element or connection type number
PN	-	Parameter number
CN	-	Connection name
CPN	-	Connection port number
EN	-	Element name
EPN	-	Element port number
GPN	-	Generation parameter number
GPV	-	Generation parameter value

Note also that all characters in the command string are SEL six-bit characters stored one character per byte with leading zeros.

The correspondence between an element or connection type and its type number is given in Table 9.

Two additional generation phase commands are available to facilitate error detection. These are:

Check Network

FF	01	06	FF

Issuance of this command causes CKNETWK to be called by the generation phase interpreter.

Table 9

Type Number Assignments

Element Type	Type Number (Hexadecimal)
Queue	01
Server	02
Source	03
Exit	04

Connection Type	Type Number (Hexadecimal)
Simple Connection	81
Priority Branch	82
Random Branch	83

Dump Network Area

FF	01	07	FF

Issuance of this command causes NWKDUMP to be called by the generation phase interpreter.

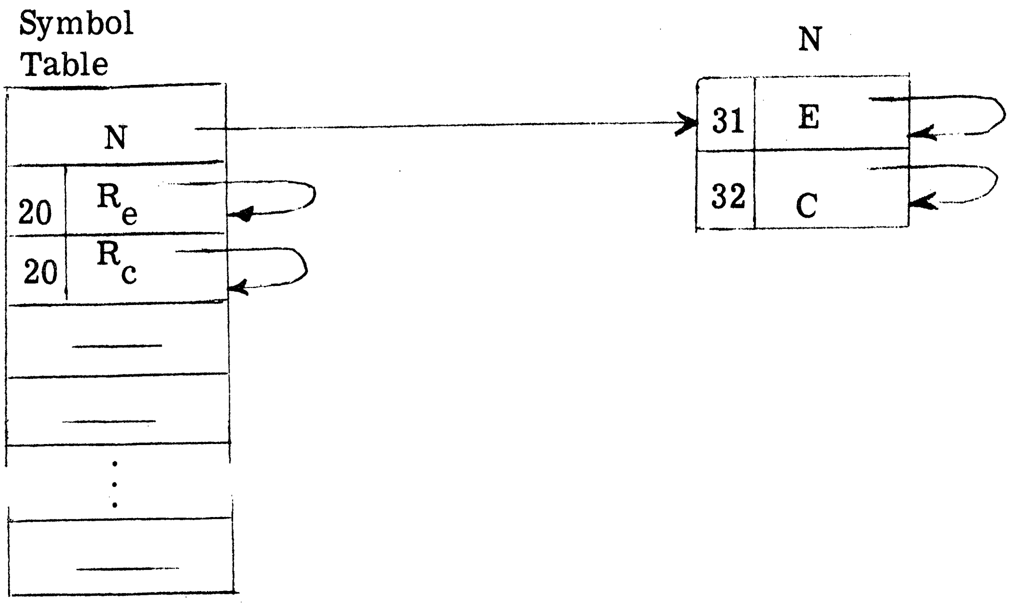
In order to clarify the effects of the above described generation phase routines, a simple example will be given illustrating each routine and showing "before and after" states of the network structure.

Assume initially that the network structure is completely empty. Schematically, the network structure will then appear as shown in Figure 1. The hexadecimal codes in the high-order bytes of pointer slots and ring heads are indicative of the information pointed to or contained in the ring. These correspondences are shown below:

Pointer and Ring Head Codes

Hexadecimal Code	Information
10	Ring of ports for element or connection.
20	Ring of unconnected element ports, R_e, or unconnected connection ports, R_c.
31	Ring of element blocks, E.
32	Ring of connection blocks, C.

A diagonal slash through the left end of a word implies a high-order byte of zero, indicating a ring element.

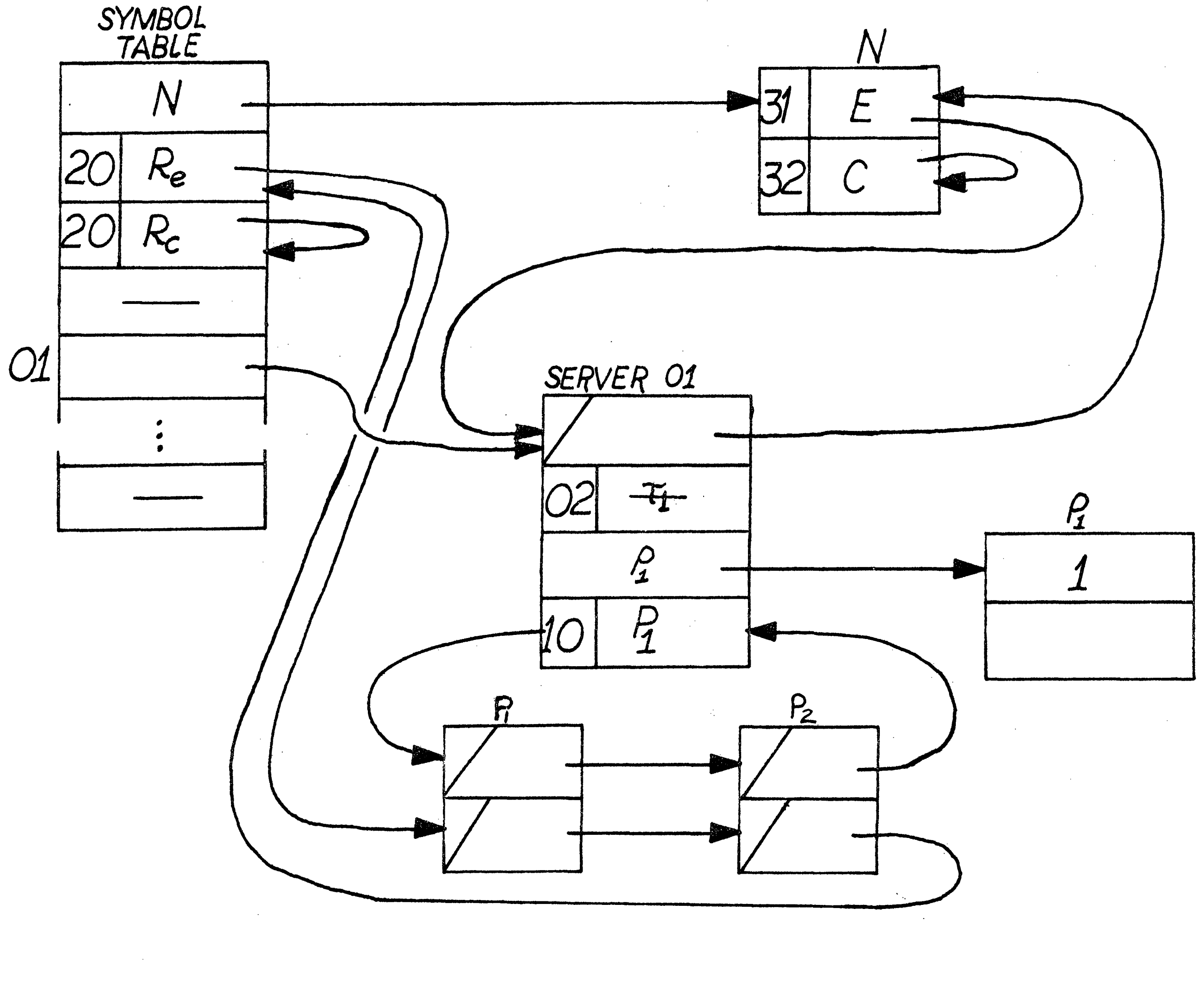


Empty Network Structure

Figure 1

Now suppose that a "Create Element or Connection" command is given indicating that a server (type 02 hexadecimal) with name 01 hexadecimal is to be "created". The network structure will then appear as shown in Figure 2. An element block corresponding to the server is inserted into the E (element) ring of the network block N and the element type 02 is inserted into the high-order byte of the τ (type) slot of the element block. Since there is as yet no type structure (this will be generated during the compilation phase of QAS), the pointer portion of the τ slot is "empty", indicated by a zero pointer. An empty parameter list block ρ_1 is created and a pointer to it is inserted into the ρ slot of the element block. The first word of the parameter list block is set equal to the number of parameters for this type of element (one in this case). The parameter mode is set to floating point. Two port blocks corresponding to the two ports of the element are inserted into the P (port) ring of the element block. These port blocks are also insertec into the unconnected element port ring, Re. A pointer to the element block is inserted into the symbol table in a slot displaced from the top of the symbol table in correspondence to the name of the element – name 01 corresponds to the first slot, name 02 corresponds to the second slot, etc. (Note that there is a zeroeth slot which should never be used.)

If at this point a "Destroy Element or Connection" command were given with element name 01 given as a parameter, the network structure would revert back to its initial state as shown in Figure 1.



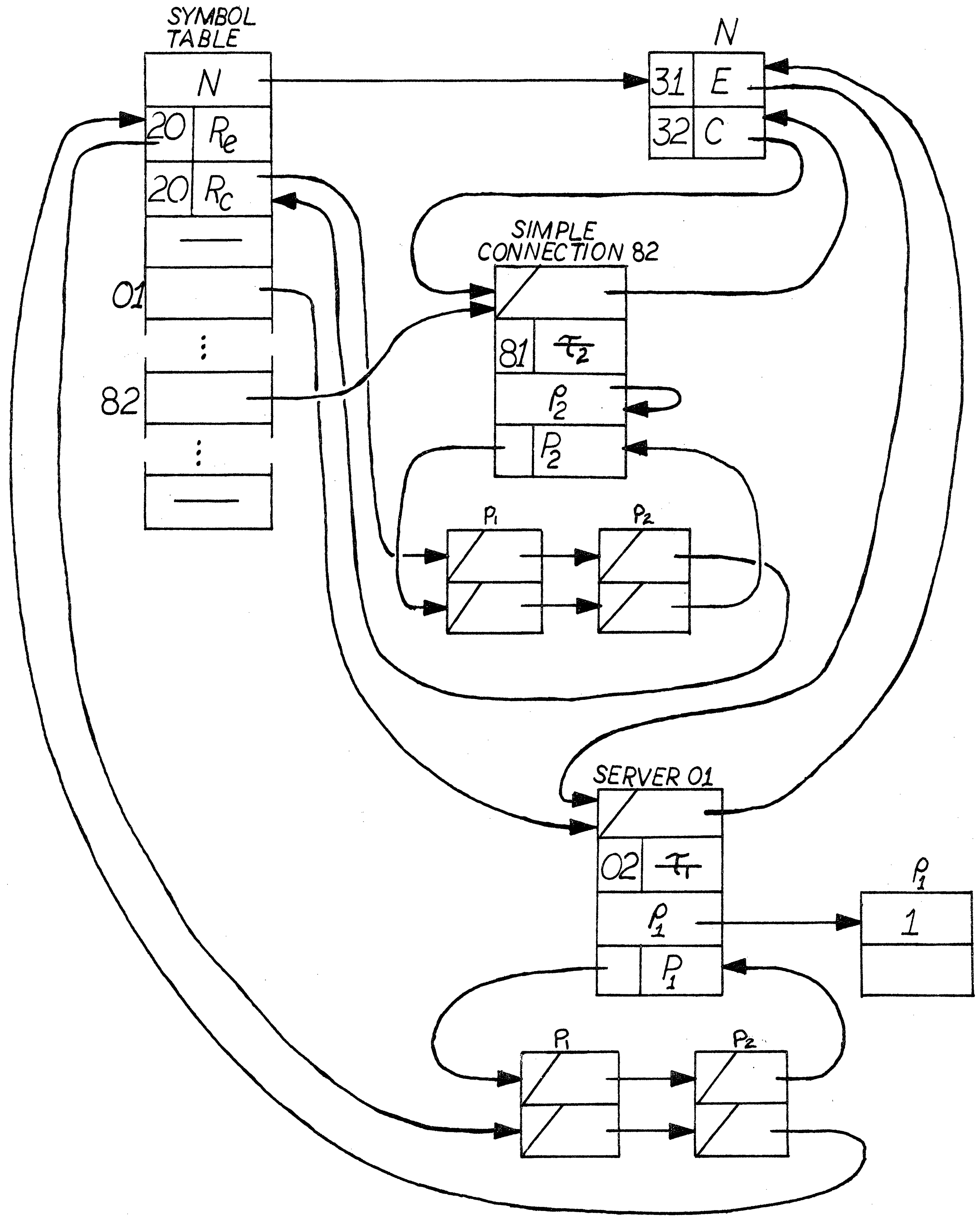
Network Structure Containing a Single Server

Figure 2

Suppose next that a "Create Element or Connection" command is given indicating that a simple connection (type 81 hexadecimal) with name 82 hexadecimal is to be "created". The network structure will then appear as shown in Figure 3. A connection block corresponding to the simple connection is inserted into the C (connection) ring of the network block N and the connection type 81 is inserted into the high order byte of the τ slot of the connection block. Since there is no type structure associated with a connection, the pointer portion of the τ slot is "empty". Since the simple connection has no parameters associated with it (for other connection types this is not necessarily so), the ρ slot of the connection block contains a pointer to itself. Two dummy port blocks corresponding to the two "ports" of the connection are inserted into the P ring of the connection block. These port blocks are also inserted into the unconnected connection port ring, R_c. A pointer to the connection block is inserted into the symbol table in the same manner as the pointer to the server element block was entered.

If at this point a "Destroy Element or Connection" command were given with connection name 82 given as a parameter, the network structure would revert back to the state it was in following the "Create Element or Connection" command for the server as shown in Figure 2.

Suppose now that an "Assign Parameter Value" command is given indicating that the value of the parameter (i.e., the mean service rate, μ) for the server is 0.5. The only change in the network structure would be to insert the value 0.5 into the empty slot in the parameter list block



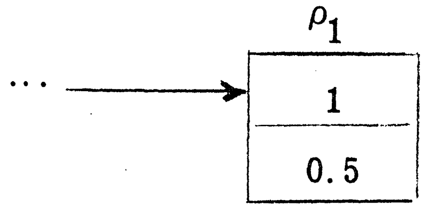
Network Structure Containing a Server and an Unconnected Simple Connection

Figure 3

ρ_1 as indicated in Figure 4. Any subsequent "Assign Parameter Value" command given for the server would merely replace the current value by the new value.

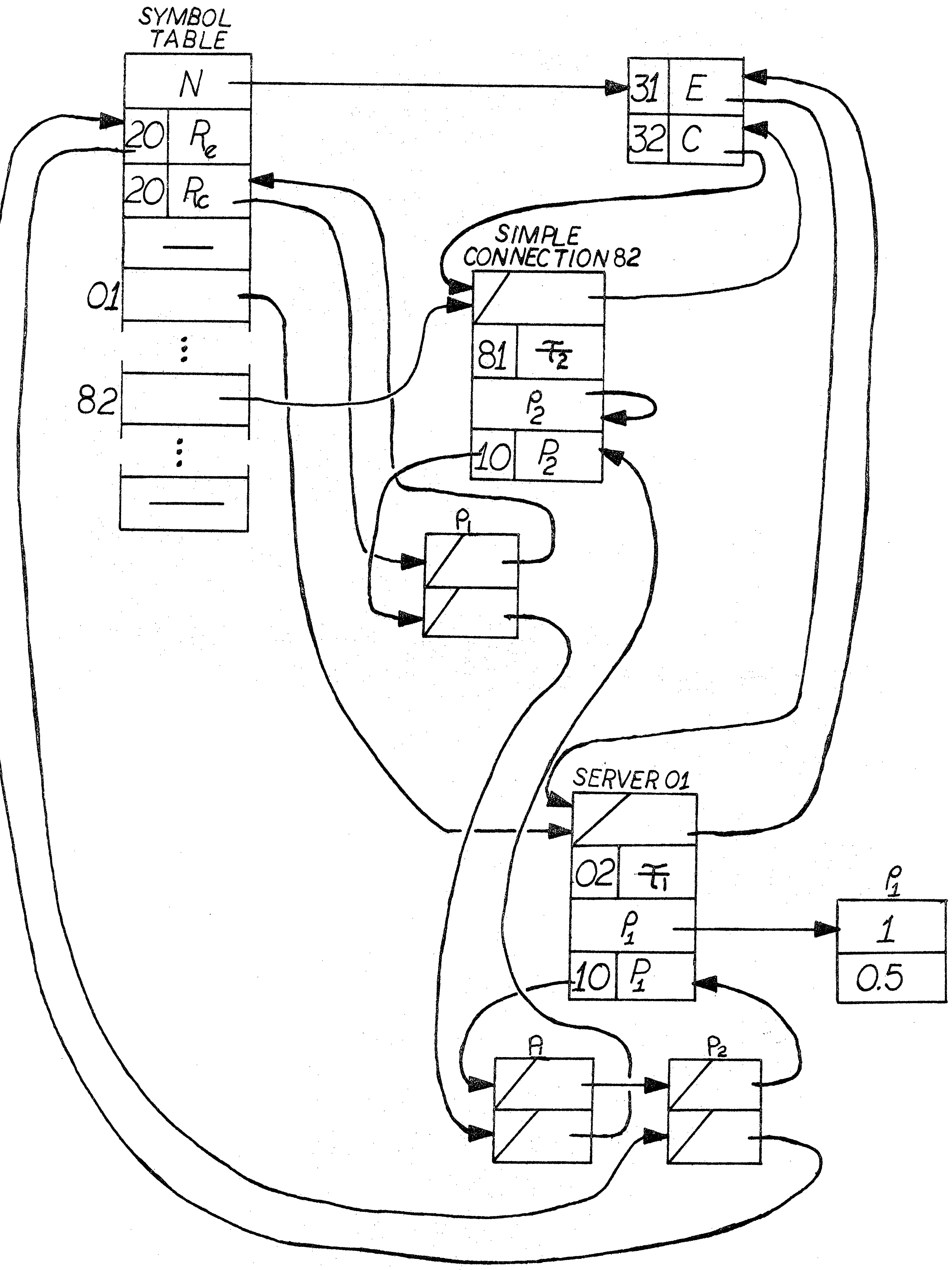
Suppose next that a "Connect" command is given indicating that port p_2 of the simple connection is to be connected to port p_1 of the server. The network structure will then appear as shown in Figure 5. Dummy port block p_2 associated with the connection is discarded and port block p_1 associated with the server takes its place in the P ring of the connection after being removed from the unconnected element port ring, Re.

If at this point a "Disconnect" command were given with element name 01 and port p_1, given as parameters, the network structure would revert back to the state it was in just before the "Connect" command, as indicated by Figures 3 and 4.



Result of Inserting a Parameter

Figure 4



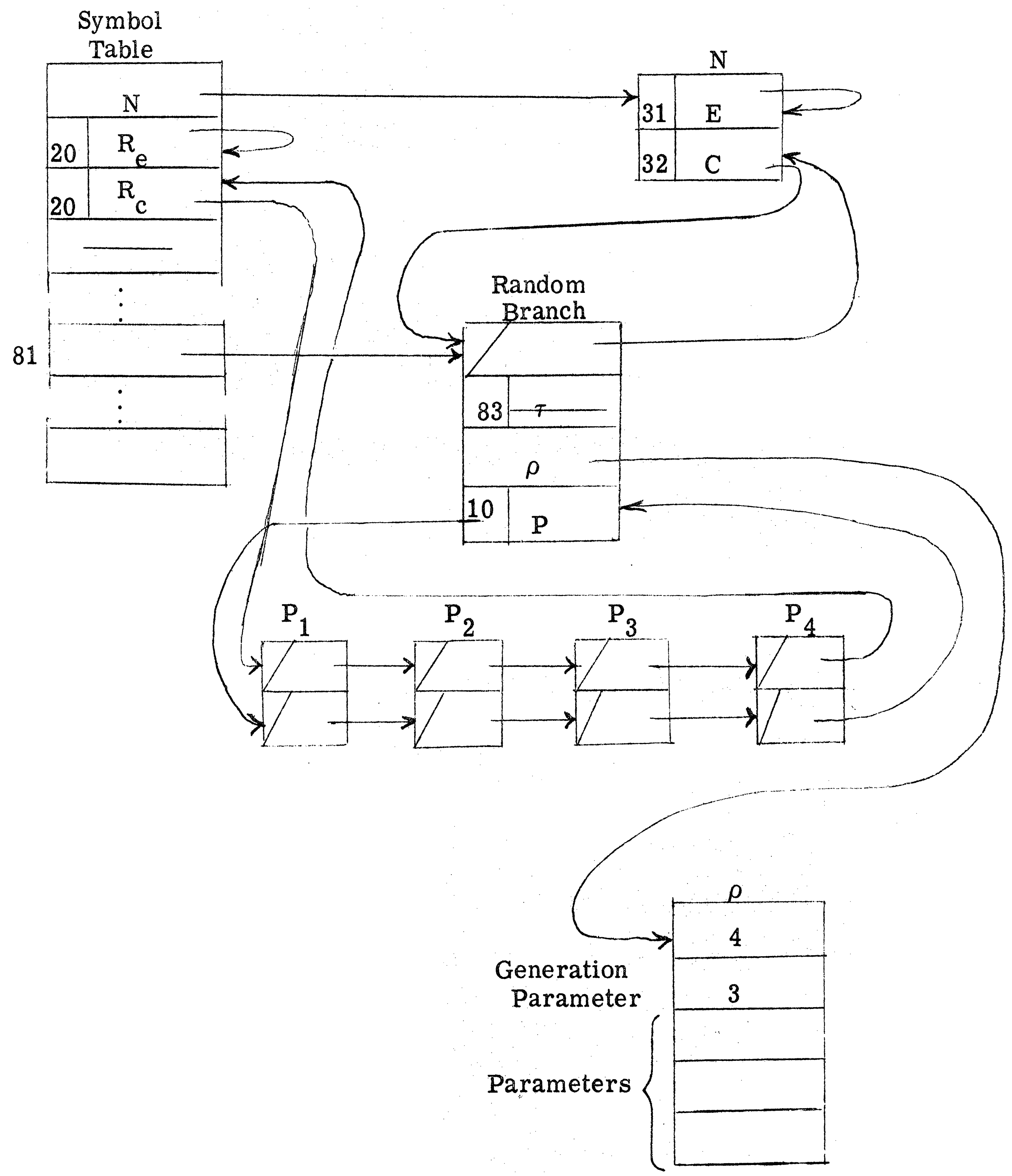
Result of a Connect Command

Figure 5

To illustrate the use of the "Alter Generation Parameter" routine a separate example will be considered.

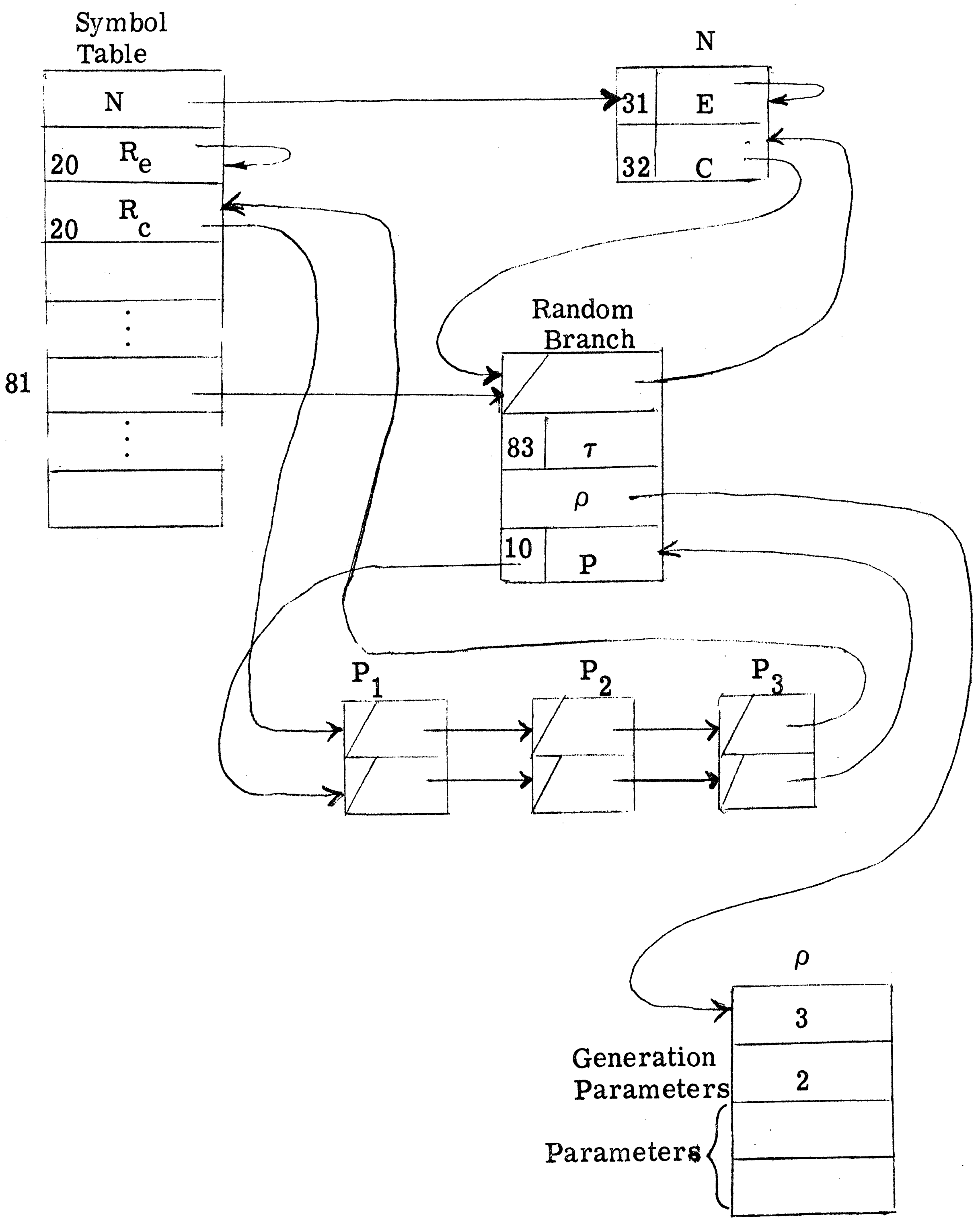
Assume initially that the network structure is empty as shown in Figure 1. Now suppose that a "Create Element or Connection" command is given indicating that a random branch (connection type 83 hexadecimal) with three branches and name 81 hexadecimal is to be "created". The command string would then contain one generation parameter having a value 3 indicating the number of branches as well as the number of parameters. The resultant network structure will then appear as shown in Figure 6.

If at this point an "Alter Generation Parameter" command is given indicating that the number of branches of random branch 81 is to be reduced to two, i.e., that the first (and only) generation parameter of this connection is to have value 2, the network structure will change to appear as shown in Figure 7. The value of the generation parameter of the random branch is changed to 2 and, correspondingly, the number of ports of the branch and the number of parameters for it are each reduced by one.



Network Structure Containing a Random Branch

Figure 6



Result of Changing a Generation Parameter

Figure 7

C. Area Dump

A routine NWKDUMP, which is used primarily for debugging purposes, may be called to obtain a hexadecimal dump of the network area on the logical I/O device SERCOM (actually, the calling sequence allows any desired QAS area to be dumped), allowing inspection of the network structure therein. To get a dump of the network area, the generation phase command X'FF0107FF' may be used.

Calling sequence:

GPR 15: Address of NWKDUMP

GPR 12: Base of area (first word in area contains its size)

D. Network Check

A routine CKNETWK may be called to check the structure resident in the network area to insure that the network represented by the structure is complete. In particular, this routine checks for missing parameters, unconnected elements or connections, and the consistency of the parameters associated with random and priority branches.

CKNETWK is called any time a command is given which leads to network compilation (compile, solve, get results, etc.). It may also be explicitly called by issuing the command X'FF0106FF'.

The order in which the various error conditions are checked is the following:

(1) Unconnected element

(2) Unconnected connection

(3) Missing parameter

(4) Sum of random branch parameters not equal to 1.0 ± 0.00001

(5) Two branches of a priority type connection have the same priority.

The first (if any) error condition detected results in the generation of a patchup request and a call to the supervisor routine PATCHUP. If no error condition is encountered, a normal return is made.

Possible patching requests which can be generated by CKNETWK are the following, where commas indicate byte boundaries and the bracket indicates fields to be filled by SELMA:

(1) Unconnected element

$$\text{FF, 01, 03, }\underbrace{\text{80, 00}}\text{, EN, EPN, FF}^{*}$$

The element name and the element port number are inserted into the patchup request in the bytes designated EN and EPN, respectively. Since there can never be a connection named X'80' nor a connection port numbered X'00', SELMA treats these two bytes as holes to be filled.

---

[*] See Section III. B for a complete description of all generation phase commands.

(2) Unconnected connection

FF, 01, 03, CN, CPN, 00, 00, FF

The connection name and the connection port number are inserted into the bytes designated CN and CPN, respectively, and the two X'00' bytes which can never be the element name or the element port number are considered the holes to be filled.

(3) Missing or inconsistent parameter

FF, 01, 02, NAME, PN, FF

The element or connection name is inserted into the byte designated NAME. The parameter number is inserted into the byte designated PN if a parameter value is unspecified (missing), and zero is inserted into this byte if all parameter values are present but something is wrong with them (currently, this implies that the sum of the parameters for a random branch does not equal $1.0 \pm 0.00001.$). A priority branch type connection, with two parameters having the same value, is treated as if one of the parameters is undefined. A simple patchup requesting that one of the parameters be redefined, is generated.

Calling sequence:

GPR 15: Address of CKNETWK

GPR 12: Base of network area

IV. Compilation Phase Routines

The procedure for compiling the network involves a successive absorption of connections. Each connection is considered in turn, and it and the elements it connects are replaced by a single equivalent element. This process is continued until no connections remain. The resulting element consists of a long list of autoevents which describe the probability intensity of every change of state which is possible.

Compilation of the network is requested by the issuance of a "compile" command or any other command which presupposes the existence of the compiled network when it does not yet exist. In the latter case, the QAS supervisor generates a compile command with default parameters.

Before the compiler is called, CKNETWK is called to insure that the network is consistent, i.e., that both rings of unconnected ports, R_e and R_c, are empty, and that all parameters are present. If this test is successful, the compiler enters a loop, successively absorbing connections until none remains.

The absorption of a connection is divided into three distinct steps. In the first step, the elements joined by the connection to be absorbed (called "associates" of the connection) are collected into a single equivalent element having ports corresponding to each of those of the associates. This collected element then replaces the associates. In the second step, the connection, which now joins only ports of a single element, is absorbed to create still another element which is equivalent to the collected element with its connection. (This two-step process eliminates the need to treat connections between ports of a single element differently from those between ports of different elements.) In the third step, the state space of the final collected element with its absorbed connection is trimmed to eliminate transient states, which contribute nothing to the solution of the network and otherwise would serve only to occupy valuable storage within the QAS data structures.

Since the compiler operates destructively upon the network structure, a copy of the network structure must be saved prior to compilation. This is done by the QAS supervisor before calling the compiler. In order to distinguish between the network area (which contains the network structure) and the workspace of the compiler (which contains the network structure when compilation begins), the compiler workspace is referred to as the "working network area".

COMPSOLV

COMPSOLV is called by the QAS supervisor to carry out the compilation process. The following steps are performed.

(I) For each connection on the network connection ring C, COMPSOLV initiates each of the three steps described above - collect associates, absorb connection and trim state set - in that order. In the case of a priority branch type connection, before initiating the above three steps subroutine SETPRIOR is called to reorder the connection port ring according to the priority parameters.

(II) After all connections have been absorbed, a dump of the working network area is given, if requested.

(III) The minor area manager is initialized by a call to RESETMIN.

(IV) A check to insure that one and only one element remains in the network is made and then the results area is acquired. Parameters for the mapping of the multi-dimensional cartesian product state space into a linear index are inserted into the results area via a call to GETRA.

(V) The working probability vector is acquired and parameters for SOLVE are extracted from the command string. SOLVE is then called to produce the steady state probability vector.

(VI) The steady state probability vector is inserted into the results area and the storage for the working probability vector area is released via a call to PUTPRVEC.

On return from COMPSOLV, the supervisor releases the working network area. The results area now contains the solution of the model. In step IV if the compiled network does not consist of exactly one element, the QAS error routine DAMMIT is called with error code X'00000D00'.

Calling sequence:

GPR 15: Address of COMPSOLV

GPR 1: Address of first byte of compile command (exclusive of the delimiting byte)

GPR 12: Base of working network area.

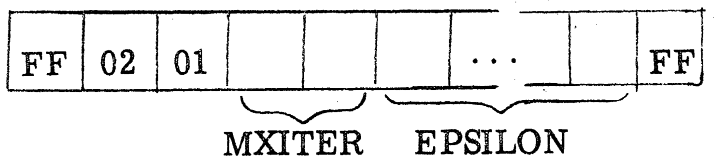
Two possible compile commands may be given; they are

(1) Compile, dump compiled network, and solve (calculate equilibrium probabilities).

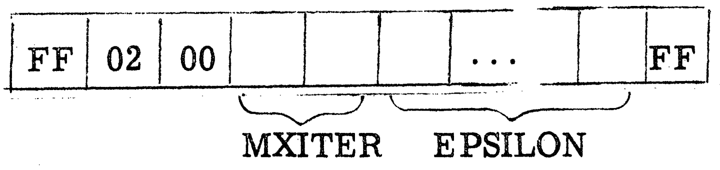
(2) Compile and solve.

The hexadecimal equivalents for these commands are:

(1)



(2)

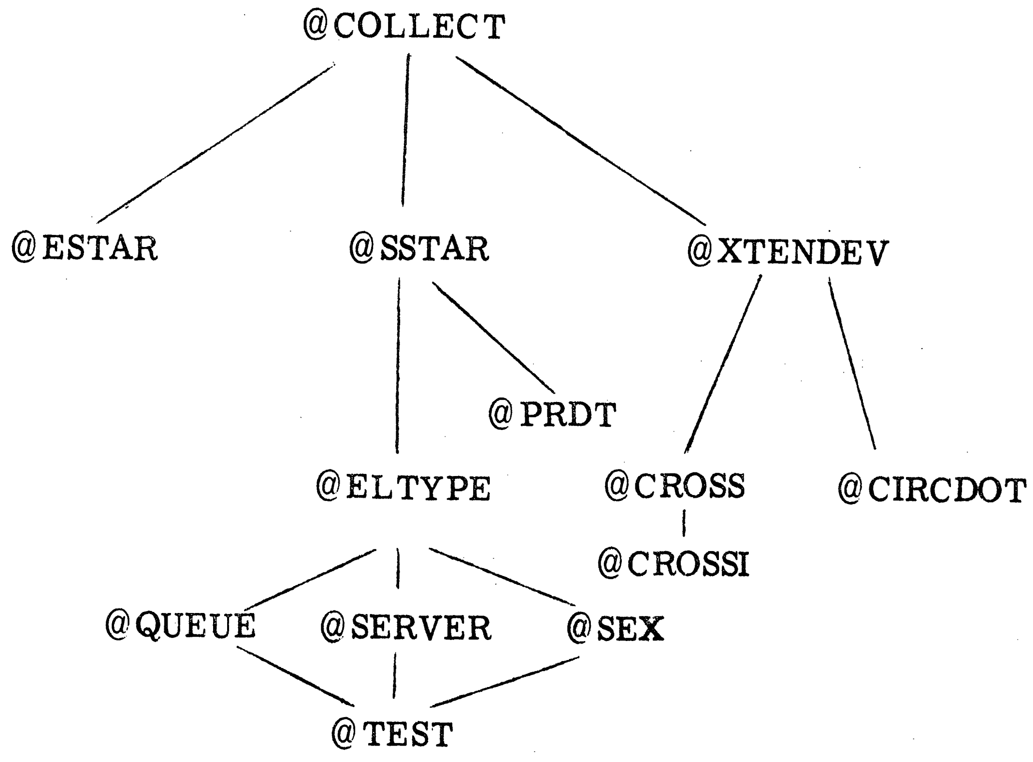


respectively, where MXITER is a halfword integer specifying the maximum number of iterations to be used in solving for the equilibrium probabilities and EPSILON is a string of SEL six-bit coded characters to be assembled into a constant and used as a convergence criterion in solving for the equilibrium probabilities.

The difference between the operations initiated by these two commands should be obvious.

A. Collect Associates

The collect associates phase of compilation is effected by a number of routines which are related to one another in a hierarchy as illustrated schematically below:



The following descriptions of each of these routines should help to clarify these relationships.

<u>@COLLECT</u>

@COLLECT collects all elements connected by a given connection and combines their type structures into a structure suitable for presentation to the connection absorption routine.

@COLLECT first calls @ESTAR which returns with all the connections associates removed from the element ring, E, and placed in a separate ring, E^*. Then it calls @SSTAR which returns with a new state set S^* which is composed of the cartesian product of all the state sets of the elements in E^*. Next it removes the first element, designated e', from E^* and inserts it into E immediately after the head. The parameter pointer ρ of e' is made to point to itself (indicating that there are no longer any parameters associated with the element) and the state set S^* becomes the state set of the element e'. Then @XTENDEV is called to modify the autoevents Ξ and the exoevents $\mathcal{Z}$ of e'. Finally, @COLLECT sequences through the remaining elements in E^*, performing the following operations on each element:

(a) The autoevents Ξ and the exoevents $\mathcal{Z}$ of the element are modified by calling @XTENDEV.

(b) The port ring P of the element is added to the beginning of the port ring of e'.

(c) The autoevent ring Ξ of the element is added to the beginning of the autoevent ring of e'.

(d) The exoevent ring $\mathcal{Z}$ of the element is added to the beginning of the exoevent ring of e'.

(e) The type structure τ of the element is destroyed.

(f) The element block itself is destroyed.

Note that the ports, the autoevents, and the exoevents are placed in an order that will match the order in which the states in S^* were combined. When the above sequence of operations has been completed for each element in E^*, the head of E^* is freed and return is made.

Calling sequence:

GPR 15: Address of @COLLECT

GPR 1: Pointer to a connection block.

GPR 12: Base of working network area.

@ESTAR

@ESTAR creates a ring E^* containing as members all elements which are associates of a given connection. First, these elements are flagged by traversing the port ring P of the connection block and following each port to its corresponding element block. After the elements have been flagged, the network element ring E is traversed, removing all flagged elements from the ring. As each flagged element is encountered, the flag is removed and the element is added to E^*. Finally, @GET is called to obtain one word of storage to act as the head of E^*.

Calling sequence:

GPR 15: Address of @ESTAR

GPR 1: Pointer to a connection block.

GPR 12: Base of working network area.

Upon return GPR 1 contains a pointer to E^*.

@SSTAR

@SSTAR forms the composite state set S* of the elements in E*. First, E* is traversed and for each element in E* the following operations are performed:

(a) @ELTYPE is called to generate the type structure τ for the element.

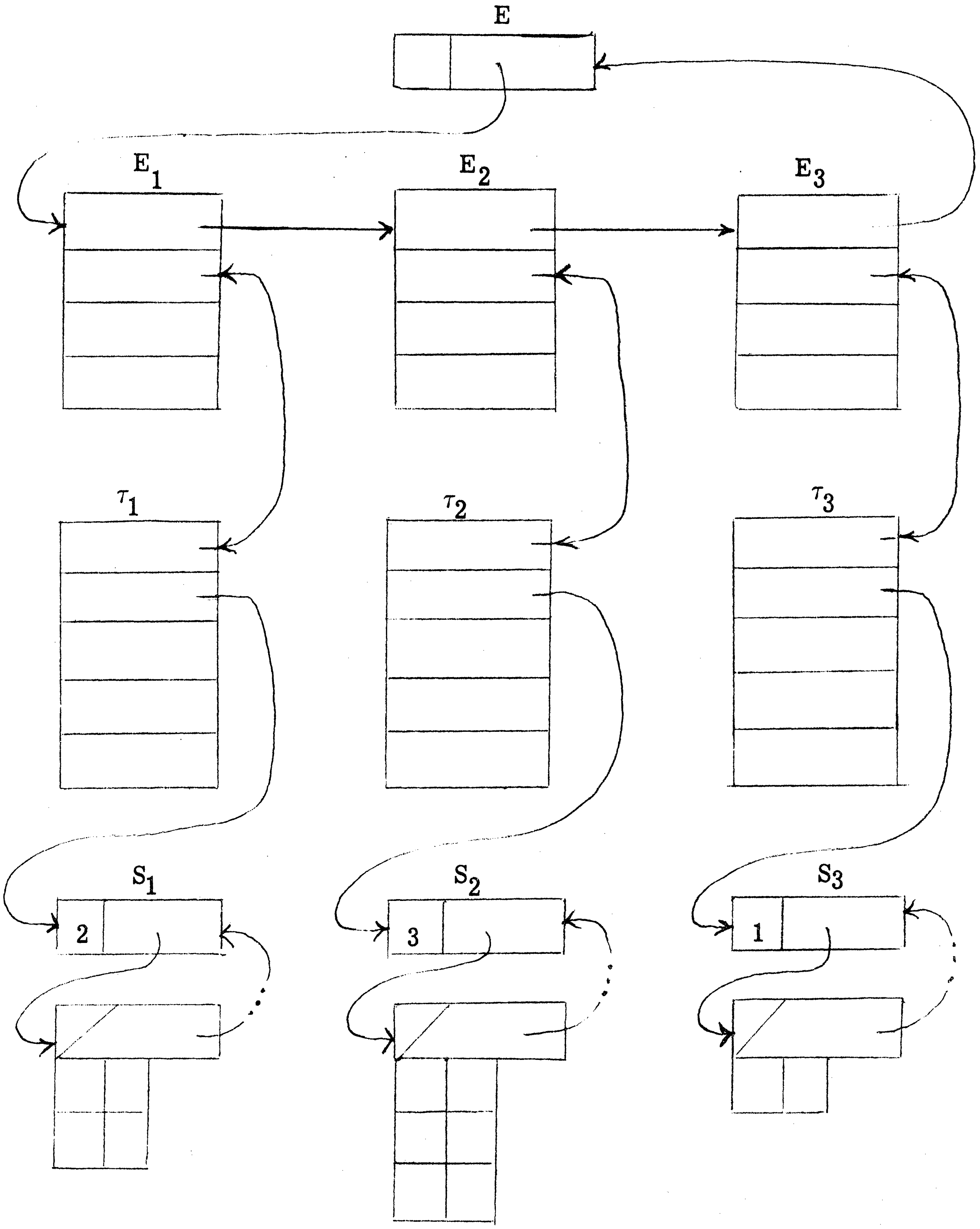
(b) The parameter list of the element is destroyed. (Parameter values are no longer needed once the type structure has been generated.)

(c) The dimension of the state set of the element is added to a running total.

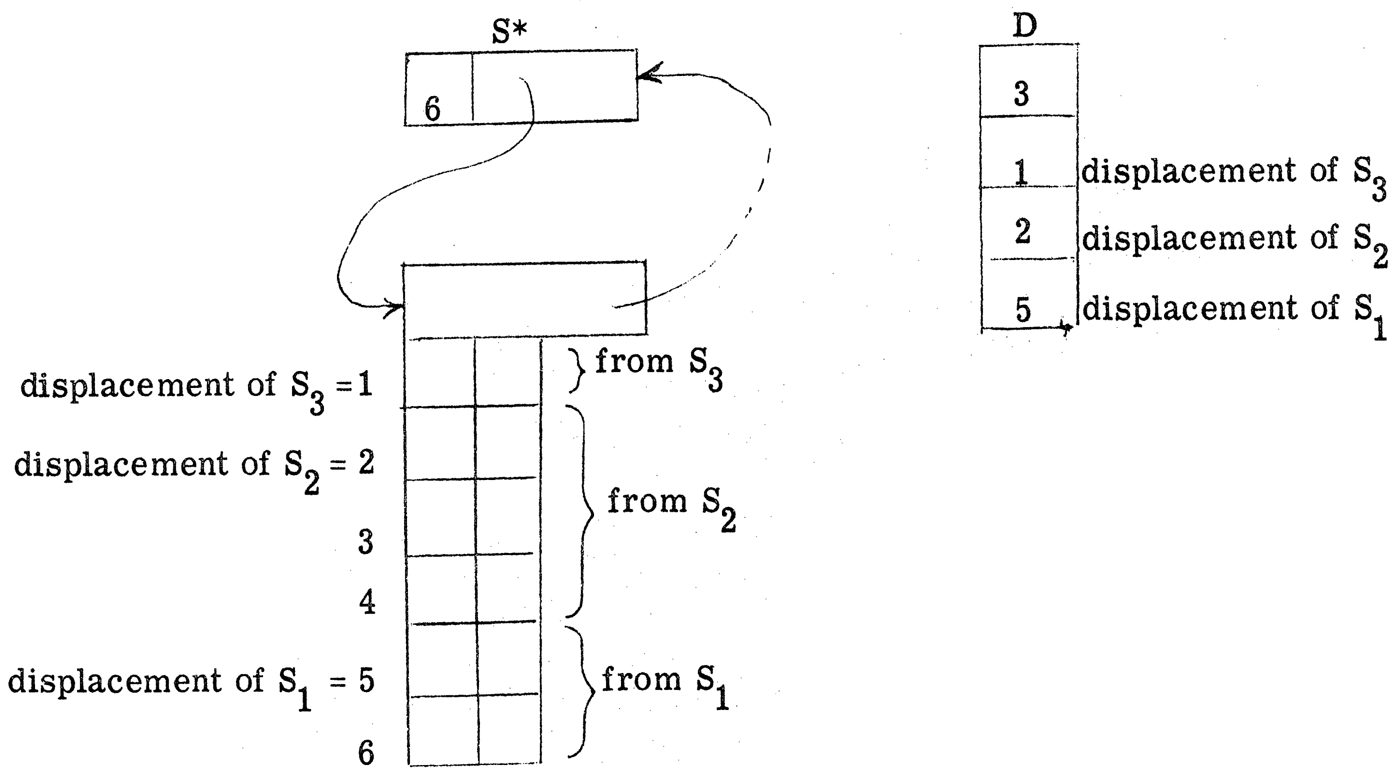
Then E* is traversed again, calling @PRDT to form the cartesian product of the state sets of the elements in E^* to yield the composite state set S^* and forming a displacement list D. The displacement list is a string of bytes beginning on a full word boundary, the first byte of which contains the number of bytes following it in the list (the number of elements in E*) and the remaining bytes of which contain the displacement of each element's contribution to the newly formed state set S*.

It is important to note that the cartesian product of the element state sets is formed backwards relative to the ordering of the elements around E*. That is, the state set of the first element of E* will be the last set of entries in an element of S*. Displacements are inserted into the displacement list from the bottom up. Thus, the ordering of D is the same as the ordering of S*.

For example, consider the following E* ring:


E
E_1
E_2
E_3
τ_1
τ_2
τ_3
S_1
S_2
S_3
2
3
1

The resultant state set S* and displacement list D would appear as follows:



Calling sequence:

GPR 15: Address of @SSTAR

GPR 1: Pointer to E* ring

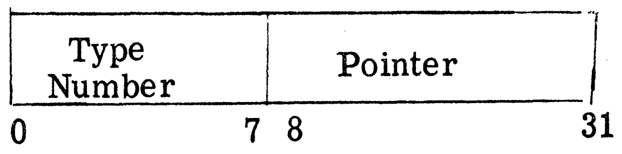
GPR 12: Base of working network area.

Upon return GPR 1 contains a pointer to S* and GPR 2 contains a pointer to D.

@ELTYPE

@ELTYPE supplies @SSTAR with a pointer to the type structure of a given element and causes the type structure to be generated if necessary.

The following convention is assumed for the type (τ) slot of an element block. (See the descriptions of the generation phase routines for the element block format.) The τ-slot occupies one full word, the high-order byte of which contains a type number and the low-order bytes of which contain a pointer to a type structure, as shown in Figure 8.



τ - slot Format

Figure 8

The correspondence between element types and type numbers is shown in Table 10. A pointer of zero indicates that a type structure has not yet been created for the element in question.

Table 10

Element Type Numbers

Element Type	Type Number (Hexadecimal)
Queue	01
Server	02
Source	03
Exit	04

When the compiler itself creates an element, it also generates the type structure for the element and hence the pointer in the τ-slot of the element block will be nonzero (i.e., will contain a pointer to the type structure). On the other hand when the element is created by the "create element" routine during generation phase, no type structure is supplied for the element and the pointer in the τ-slot of the element block will be zero. In this case the type structure is supplied by @ELTYPE when it is needed by the compiler. @SSTAR requests a pointer to the type structure of an element through @ELTYPE. When an element is specified, if the type structure is already present, a pointer to the type block is immediately returned. Otherwise, if the type structure is not yet present, @ELTYPE calls a type-evaluation routine which creates the type structure for the element using the parameter values specified for that element. The correspondence between element types and type-evaluation routines is shown in Table 11.

Table 11

Type-Evaluation Routines

Element Type	Evaluation Routine
Queue	@QUEUE
Server	@SERVER
Source, Exit	@SEX

Since the type structure for an element is dependent upon the parameter values associated with the element, the type-evaluation routines call upon a routine called @TEST which checks the parameter list corresponding to the element in question to insure that values have been specified for all parameters.

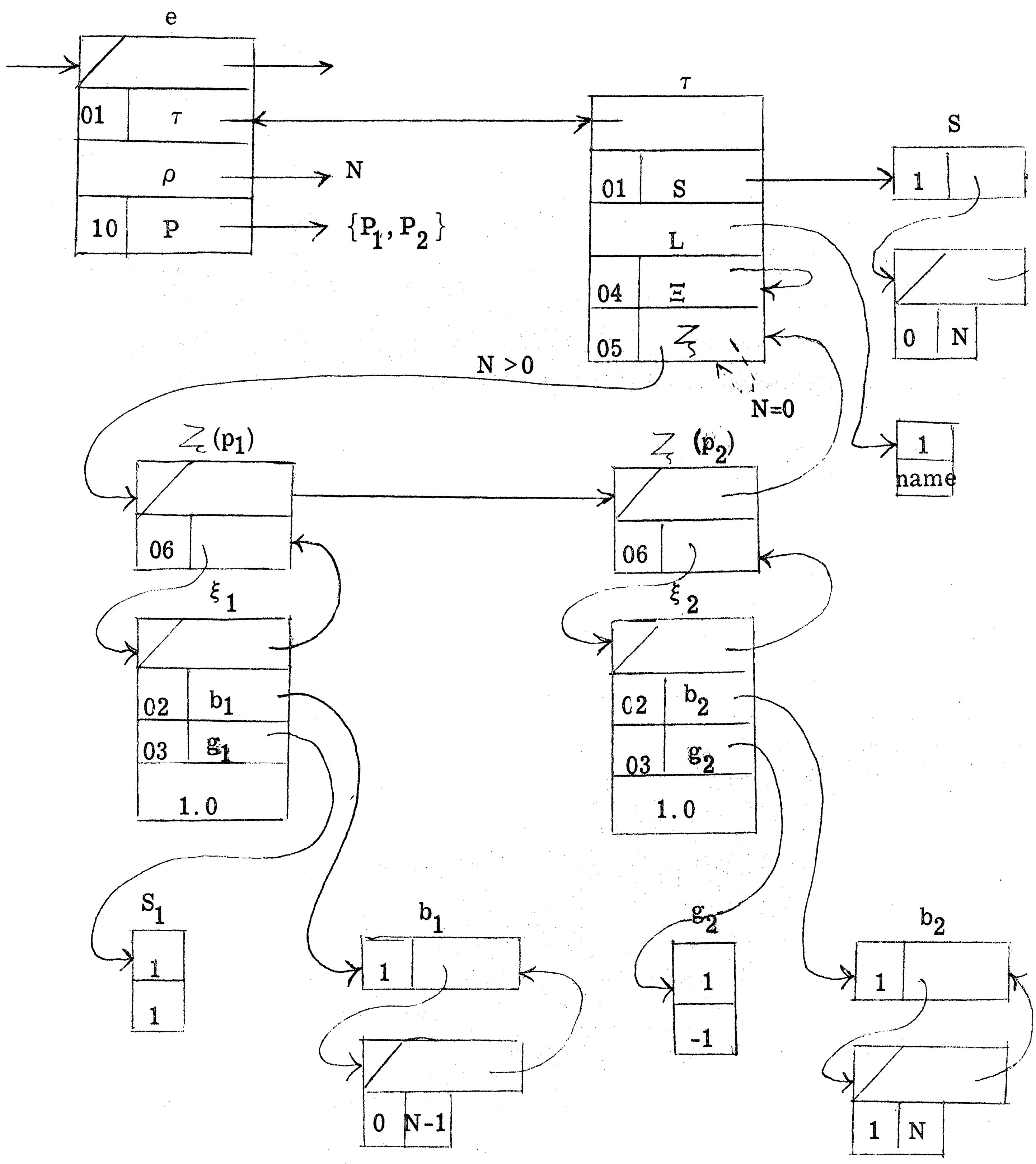
The type structures generated by the various type-evaluation routines are illustrated in Figures 9, 10, and 11. The hexadecimal codes in the high-order bytes of pointer slots and ring heads are indicative of the information pointed to or contained in the ring. These correspondances are shown in Table 12.

Table 12
Pointer and Ring Head Codes

Hexadecimal Code	Information
01	State set
02	b ring
03	g vector
04	Autoevents, Ξ
05	Exoevents, Z_τ
06	Exoevents corresponding to port p_i, $Z_\tau(p_i)$

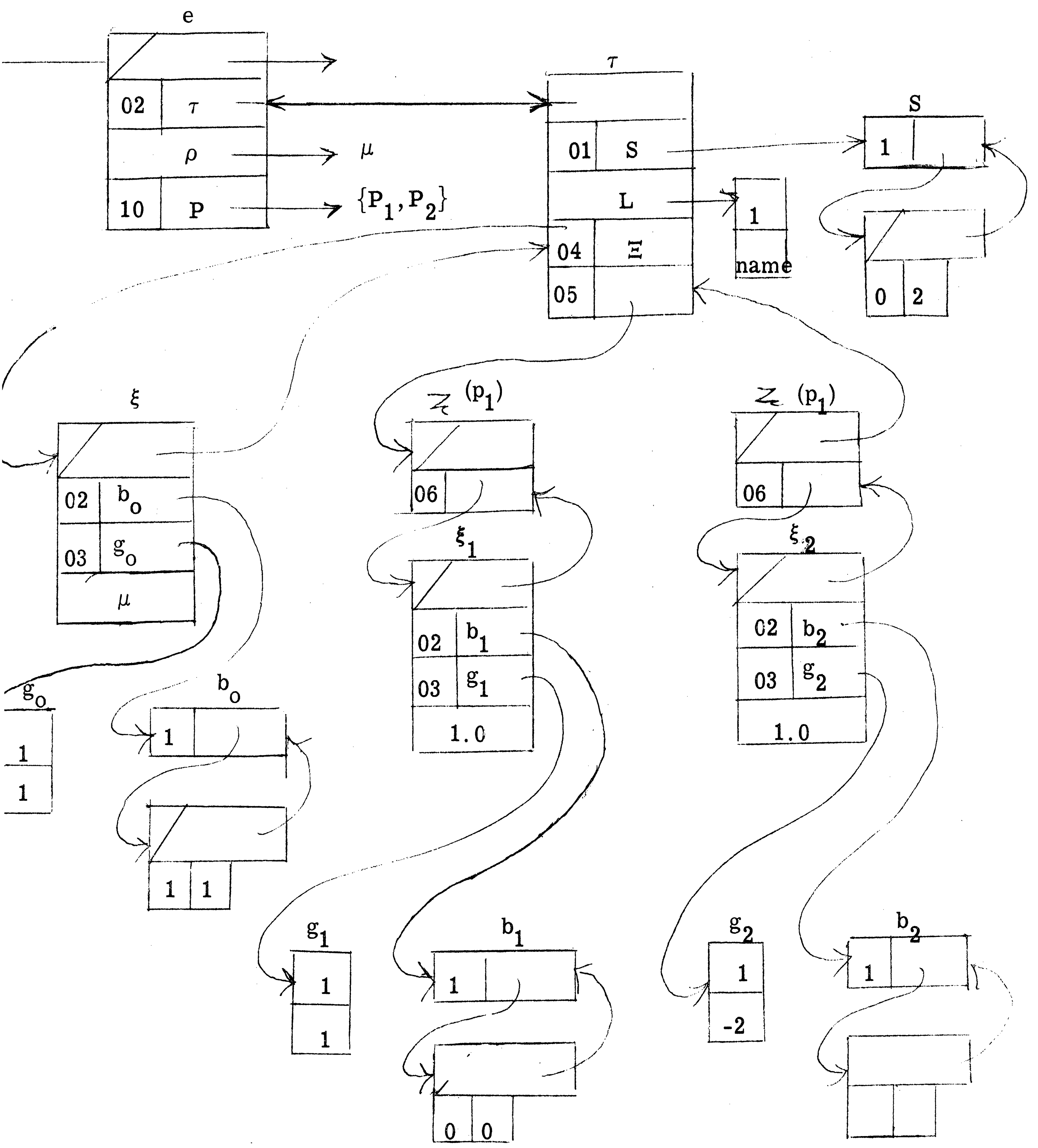
The first slot of the type block τ contains a pointer back to the τ-slot of the element block. The second slot contains a pointer to the state set structure for the element in question. This structure has the format described for sets in the description of the set manipulation routines.

The third slot of the τ-block is used for state mapping purposes. This slot contains a pointer L to a block of n+1 bytes called the name stack, the first byte of which contains a count n of the entries in the block and the remaining n bytes of which contain element names, as shown in the following diagram:



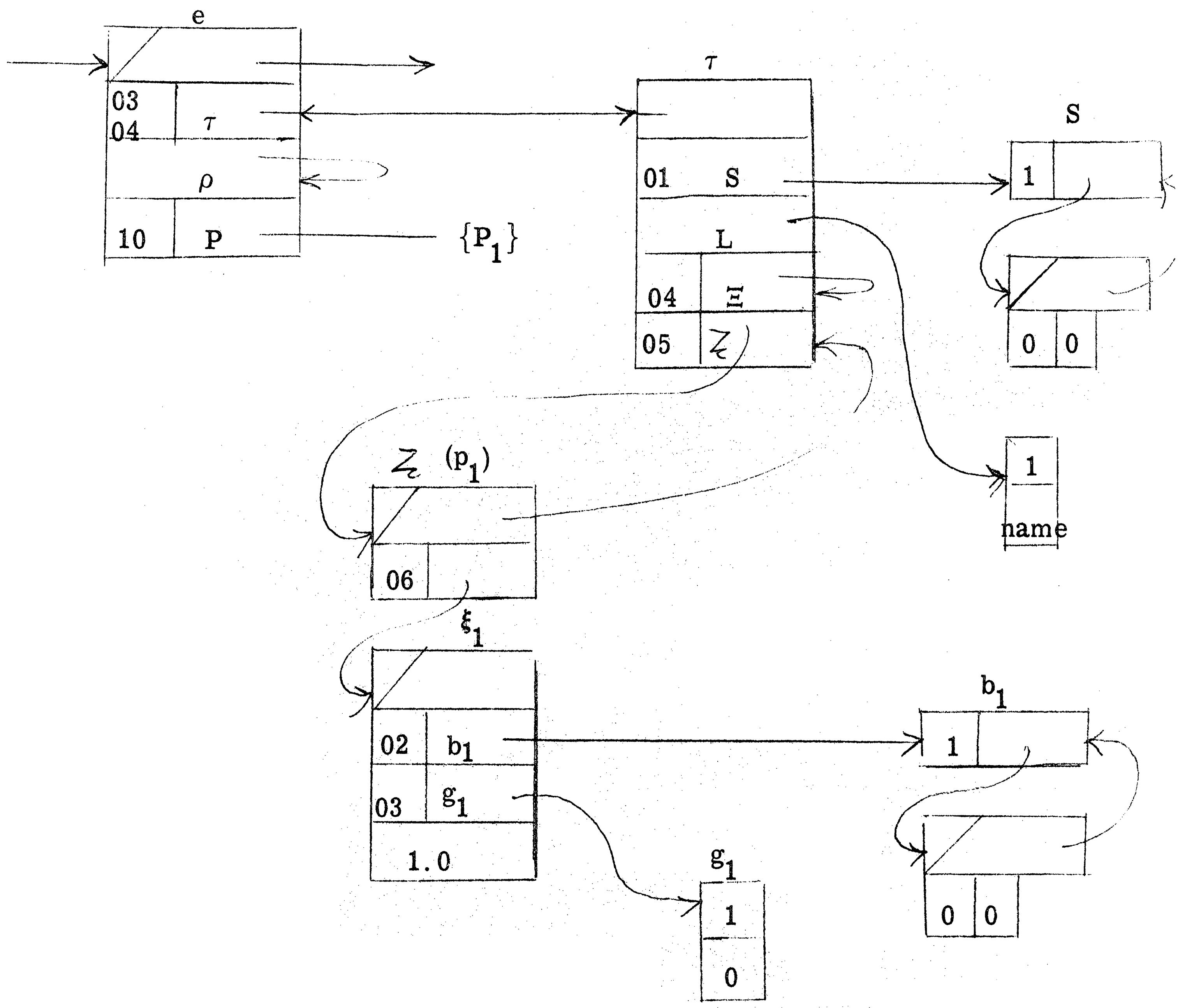
Type Structure for Queue

Figure 9



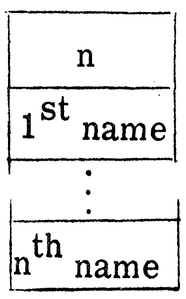
Type Structure for Server

Figure 10



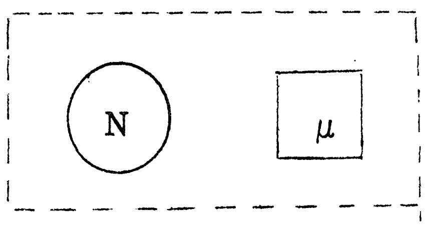
Type Structure for Source-Exit

Figure 11

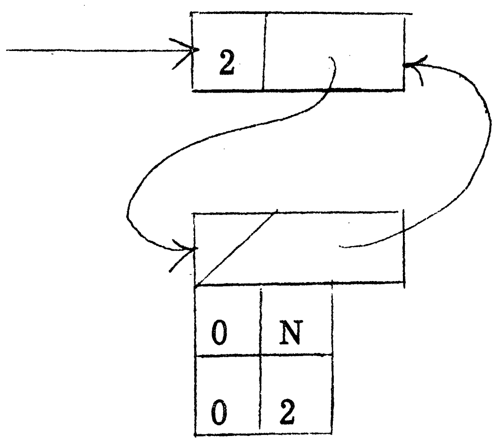


The sequence of the names in the name stack corresponds to the ordering of the contributions made by the respective component elements to the state set S and to the b rings of the composite element which the τ-block represents.

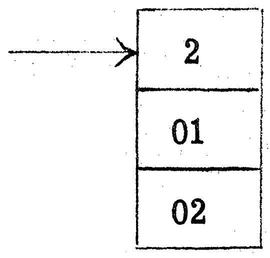
For instance, if the composite element were the following:



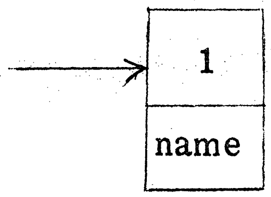
which would yield a state set of the form



and assuming that the queue had name 01 and the server had name 02, then the name stack would appear as follows:



For each simple element (queue, server, source, or exit) the name stack appears as follows:



and it is to this form that all name stacks are initially set. Only after at least some amount of compilation do any name stacks assume the more general form.

Currently @ELTYPE initializes the name stack for each simple element when it is called upon to create the type structure for the element, and during compilation the name stacks are concatenated to yield the final name stack which supplies necessary state mapping information.

The fourth slot of the τ-block acts as the ring head for the ring of autoevents for the element and the fifth slot acts as the ring head for the ring of exoevents. Actually, the ring of exoevents is a ring of rings of exoevents, the upper ring serving to segregate the exoevents into sets of exoevents corresponding to each port of the element.

The format of an event block (both for autoevents and exoevents) is the following: The first slot of an event block serves as a ring link. The second slot contains a pointer to a set structure (identical in format to the state set structure) designated a b-set which indicates for which states the event can occur. The third slot contains a pointer to a vector (having a format like that of the vector in the description of the set manipulation routines) designated a g-vector which when added to the b-set yields the set of states to which the event causes a transition. The fourth slot contains a transition intensity for the event.

Calling sequence:

GPR 15: Address of @ELTYPE

GPR 1: Pointer to element block

GPR 12: Base of working network area.

Upon return GPR 1 contains a pointer to the type block of the element. The QAS error routine DAMMIT is called with error code X'01000000' if there is no element type which corresponds to the type code given in the high-order byte of the τ-slot of the element block.

@QUEUE

@QUEUE calls @TEST to insure that a value has been specified for the parameter of the queue in question and then generates the appropriate type structure.

Calling sequence:

GPR 15: Address of @QUEUE.

GPR 1: Pointer to parameter block of given queue.

GPR 12: Base of working network area.

Upon return GPR 1 contains a pointer to the type structure generated.

@SERVER

@SERVER calls @TEST to insure that a value has been specified for the parameter of the server in question and then generates the appropriate type structure.

Calling sequence:

GPR 15: Address of @SERVER

GPR 1: Pointer to parameter block of given server.

GPR 12: Base of working network area.

Upon return GPR 1 contains a pointer to the type structure generated.

@SEX

@SEX generates the type structure for the source or the exit in question.

Calling sequence:

GPR 15: Address of @SEX.

GPR 12: Base of working network area.

Upon return GPR 1 contains a pointer to the type structure generated.

@TEST

@TEST checks the parameter list indicated to determine whether all parameter values have been specified. A parameter value is considered to be unspecified if either of the following codes is found:

X'000E0000' - unevaluated fixed point

X'FE0F0000' - unevaluated floating point

The QAS error routine DAMMIT is called with error code X'02000000' if any parameter value is found to be unspecified.

Calling sequence:

GPR 15: Address of @TEST.

GPR 1: Pointer to parameter block of given element.

GPR 12: Base of working network area.

@XTENDEV

The purpose of @XTENDEV is to extend the b-sets and the g-vectors of the events of a given associate of a connection to include the events' effects within the new state set S* of the collected element. @XTENDEV is given a pointer to the associate's τ-block and a displacement (taken from the displacement list D created by @SSTAR) indicating where in the state set S* the element makes its contribution. It then takes each b-set ring of the element's events (both autoevents and exoevents) and calls @CROSS to suitably modify the b-set. Similarly, it takes each g-vector of the element's events and calls @CIRCDOT to suitably modify the g-vector. The newly formed b-sets and g-vectors are then made to replace the old b-sets and g-vectors, respectively.

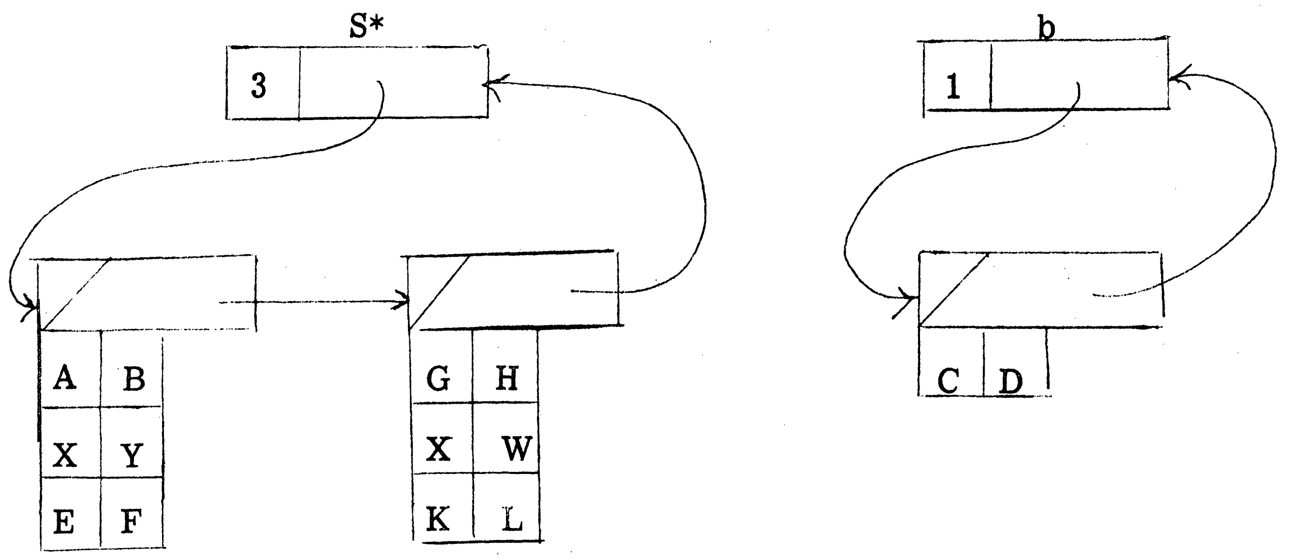
Calling sequence:

GPR 15: Address of @XTENDEV.

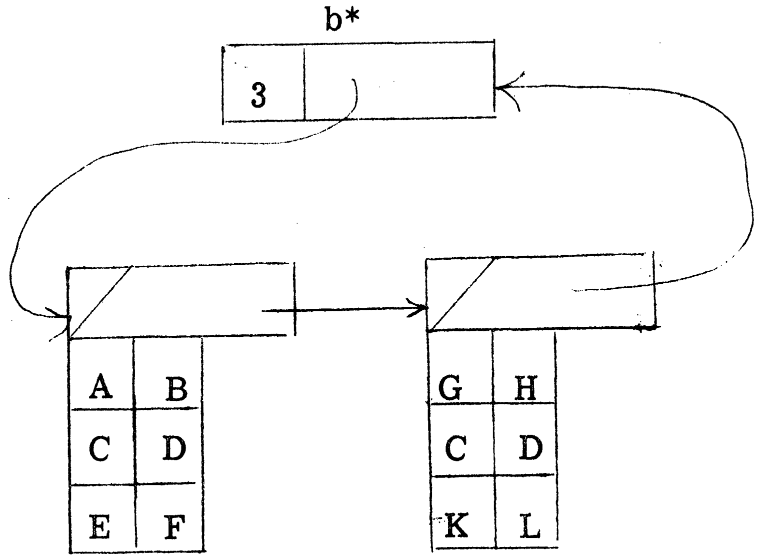
GPR 0:	Displacement.
GPR 1:	Pointer to τ-block of associate.
GPR 2:	Pointer to S*.
GPR 12:	Base of working network area.

@CROSS

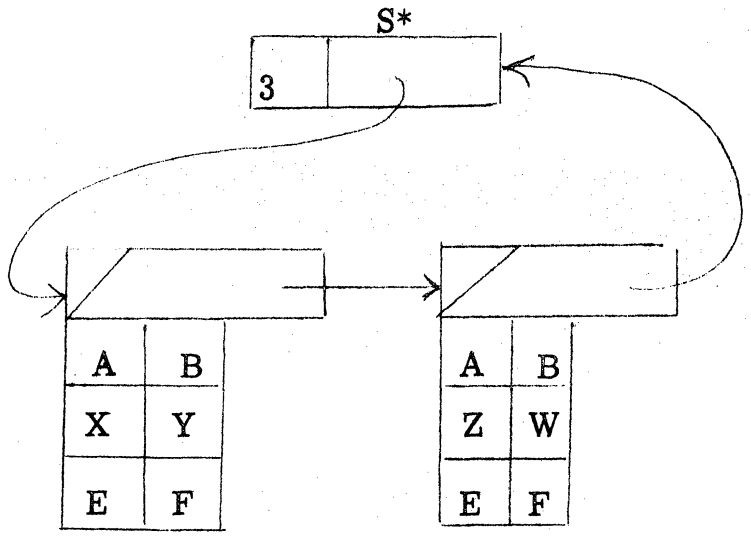
The function of @CROSS is to suitably modify S* (for the purposes of extending events only) to eliminate the multiplicity of states which could arise within the b-sets if S* were not modified when @CROSSI is called to modify the b-sets. In order to understand just what effect @CROSS has on S* it will be necessary to first consider the operation performed by @CROSSI, which is the following: A ring of state blocks b* (which comprises a set structure in the format described earlier) is generated in the following manner. The first block within the S* ring is inserted into b* and the first block within the b-set ring is copied on top of it at the displacement given. This process is then continued, forming a new block within b* for each possible combination of blocks within S* and b. For instance suppose that S* and b were as shown below and that the displacement given were 2.



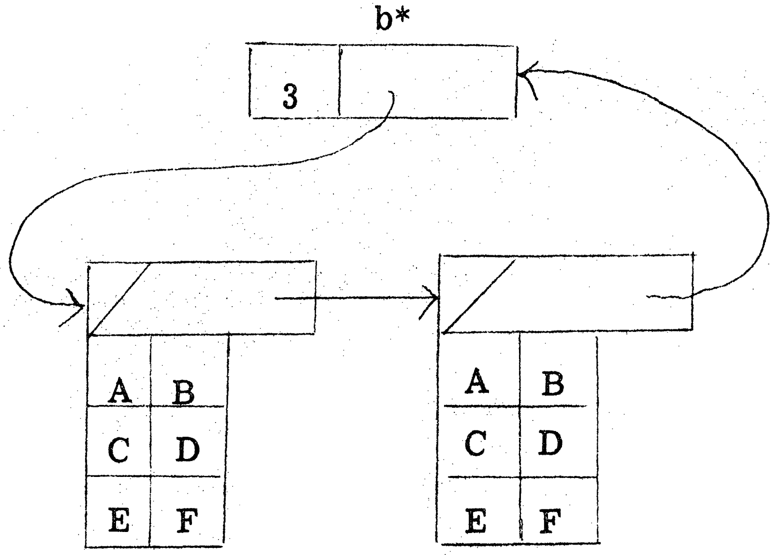
Then the resultant b* would be as follows:



The difficulty arises (and herein lies the necessity of the operation performed by @CROSS) when S* contains blocks which are identical except for that portion contributed by the element from which the b-set came. For instance, suppose that S* were as shown below with b the same as above and a displacement of 2.

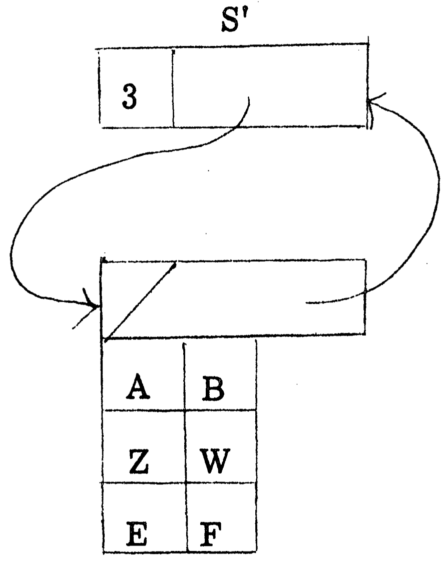


The resultant b* would then be as follows:



Clearly, one of these blocks can be eliminated. In fact unless duplicate blocks such as these are eliminated, the structure soon mushrooms tremendously resulting in a gross waste of storage and time.

Hence, @CROSS checks for instances where this multiplicity will arise and prunes the S* structure, creating a new structure S' that it passes on to @CROSSI. In the example above S* would be modified to the following:



before being passed on to @CROSSI.

Calling sequence:

GPR 15: Address of @CROSS.

GPR 0: Displacement.

GPR 1: Pointer to S*.

GPR 2: Pointer to b.

GPR 12: Base of working network area.

Upon return GPR 1 contains a pointer to the new set b*.

<u>@CROSSI</u>

@CROSSI forms a pseudo-cartesian product (as described under the @CROSS description) between two rings S* and b.

Calling sequence:

GPR 15: Address of @CROSSI

GPR 0: Displacement.

GPR 1: Pointer to (modified)S*.

GPR 2: Pointer to b.

GPR 12: Base of working network area.

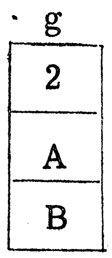
Upon return GPR 1 contains a pointer to the new set b*. If the displacement and the dimensions of S* and b are such that a block of b does not fit entirely within a block of S*, the QAS error routine DAMMIT is called with error code X'000000A0'.

Note that @CROSSI will not operate properly for dimensions in in excess of 128.

@CIRCDOT

@CIRCDOT performs the following operation, using S* and a g-vector to obtain a new vector g*. A vector of n+1 bytes (where n is the dimension of S*) is obtained. The dimension of S*, n, is placed in the first byte of the vector and the rest of the bytes are set to zero. Then the g-vector is copied on top of the newly created vector at the displacement given to yield the final vector g*.

For example, if the dimension of S* were 5, the displacement given were 2, and the g-vector appeared as follows:



then the new vector g* would appear as follows:

g*
5
0
A
B
0
0

Calling sequence:

GPR 15: Address of @CIRCDOT.

GPR 0: Displacement.

GPR 1: Pointer to S*.

GPR 2: Pointer to g.

GPR 12: Base of working network area.

Upon return GPR 1 contains a pointer to the new vector g*. If the displacement and the dimensions of S* and g are such that g does not fit entirely within g*, the QAS error routine DAMMIT is called with error code X'000000B0'.

Note that @CIRCDOT will not operate properly for dimensions in excess of 128.

@PRDT

@PRDT performs essentially the same operation as PRDT which was described under the set manipulation routines (namely, forming the cartesian product between two sets) but its calling sequence is "simplified" to make life easier for @SSTAR. Also, the storage obtained from @GET is in one contiguous block to facilitate releasing the storage. The number of words obtained is returned to the calling program so that when an intermediate result is to be destroyed, the entire block may be released with one call to @FREE rather than having to sequence through the structure freeing each fragment individually. Since @SSTAR may yield a number of intermediate results before the final cartesian product S* is obtained, this can result in a considerable time savings.

Calling sequence:

GPR 15:	Address of @PRDT.
GPR 1:	Pointer to state set A.
GPR 2:	Pointer to state set B.
GPR 12:	Base of working network area.

B. Absorb Connection

The absorption of connections is implemented via two routines ABSORB and SPON.

ABSORB

The entire absorption of a connection is accomplished by calling ABSORB with a pointer to a connection block. When called, ABSORB performs the following sequence of operations:

(1) The connection block indicated is removed from the network connection ring C.

(2) The element ports which are joined by the given connection are destroyed. The exoevents which correspond to these ports are placed on the connection's port ring (which no longer contains any ports).

(3) The minor area to be used as the spontaneous event set area is set up by a call to EXPMINOR, which is the main entry in the minor area manager. This sets up one block of storage as a minor area. (Blocks acquired for the absorption of previous connections are reused.)

(4) The entry in SPON which corresponds to the connection type being absorbed is called. SPON then generates the spontaneous event set using the exoevents on the connection's port ring and the parameters (if any) associated with the connection.

(5) The augmented spontaneous event set is created using the spontaneous event set generated by SPON and the element state set.

(6) The element state set is pruned to eliminate all forbidden states.

(7) The connection block and its associated ring of exoevents are destroyed.

(8) The autoevent set of the element and the augmented spontaneous event set are consolidated to yield a new autoevent set.

(9) The exoevent set of the element and the augmented spontaneous event set are consolidated to yield a new exoevent set.

As an illustration of the above sequence of operations, consider first the fragment of a working network structure shown in Figure 12. After step (2) the fragment will appear as shown in Figure 13. After step (5) the structure shown in Figure 14 will be present in addition to everything shown in Figure 13. After step (7) the structure pointed to by $\mathfrak{C}_k$ in Figure 13 will have been destroyed. And after step (9) the fragment will appear as shown in Figure 15.

Note that the execution of ABSORB is not dependent upon the connection type being processed. All computations unique to a connection type are performed in SPON. Connection type becomes important to ABSORB only when a particular entry to SPON is to be called. At this point an illegal type number will result in a fatal error.

The large (minor) area in which the spontaneous event sets are created is obtained by ABSORB (as indicated above) and the size of this area is currently set at 2000 words. If necessary this size can be changec by changing the value of the constant called SPONSIZE.

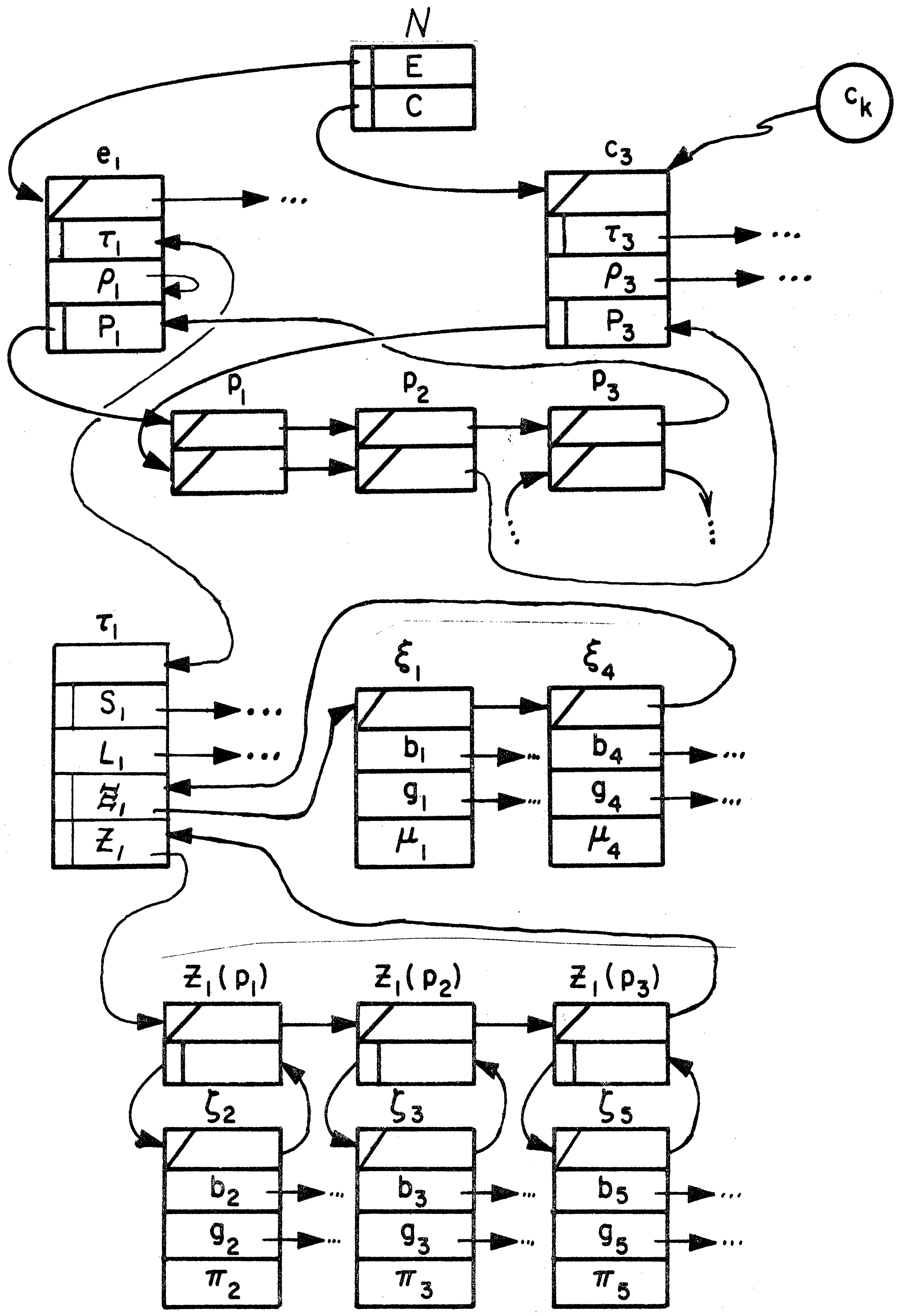
Calling sequence:

GPR 15: Address of ABSORB.

GPR 1: Pointer to connection block.

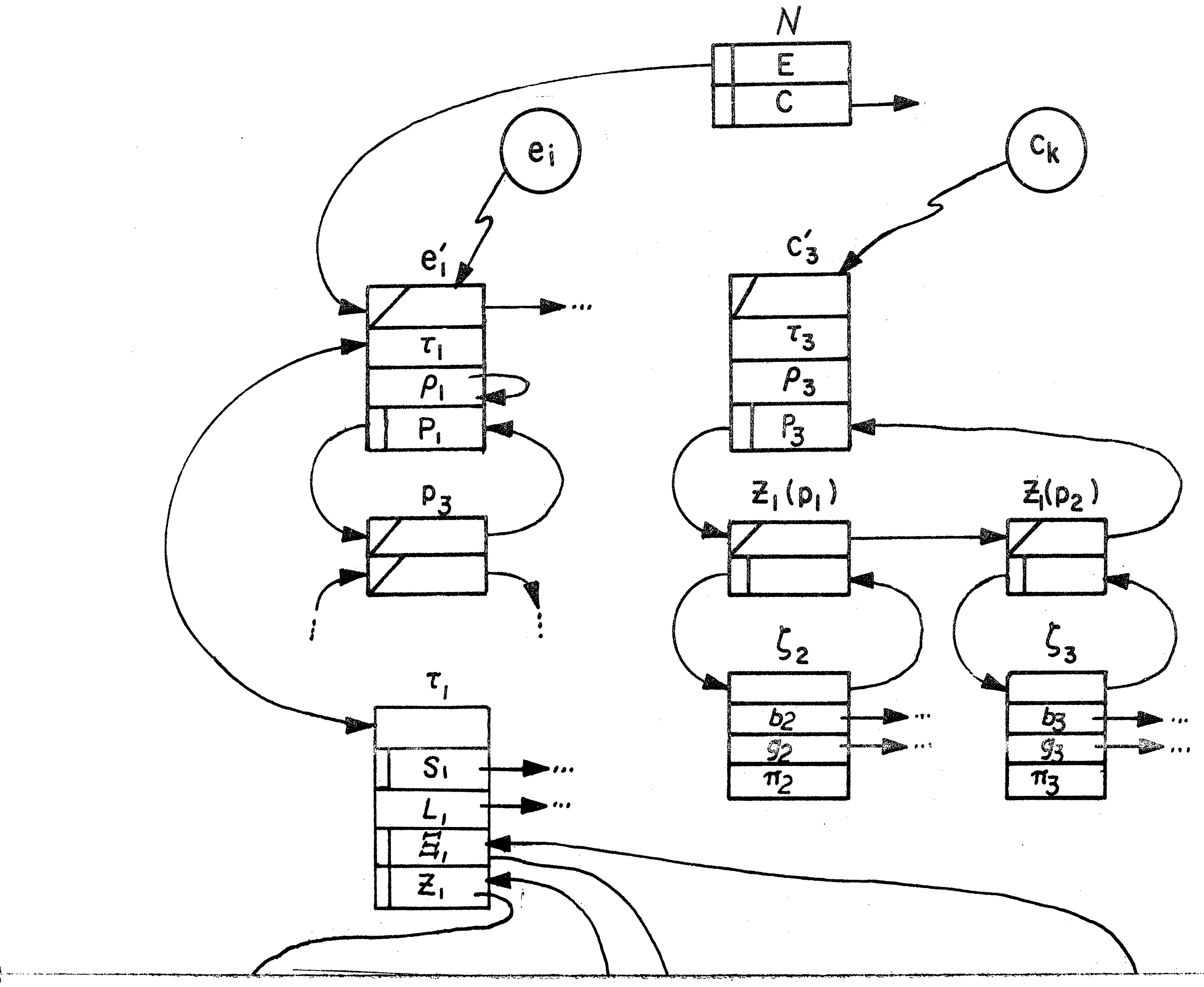
GPR 12: Base of working network area.

The QAS error routine DAMMIT is called with the indicated error code if any of the following situations arises:



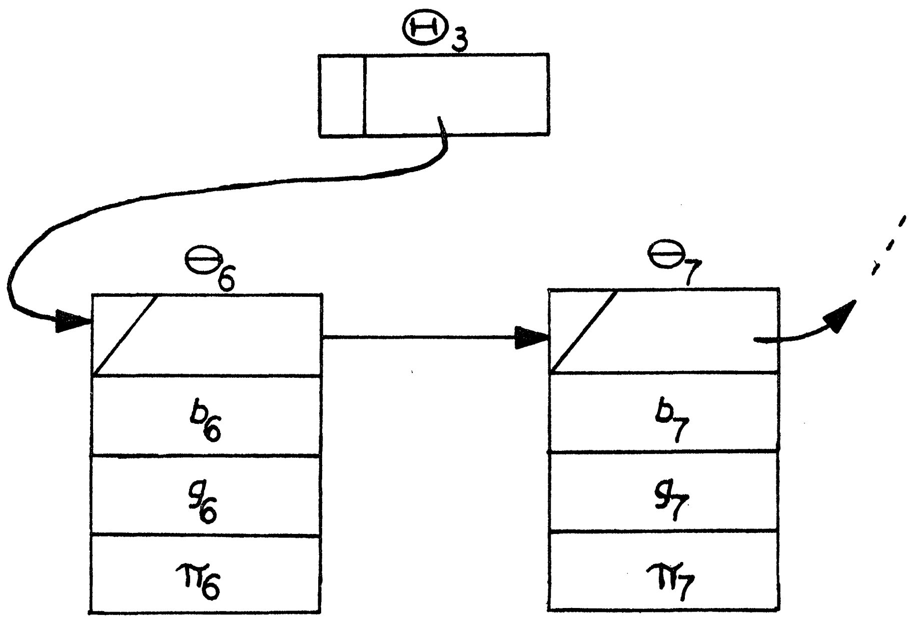
Fragment of Working Network Structure at Beginning of Connection Absorption

Figure 12



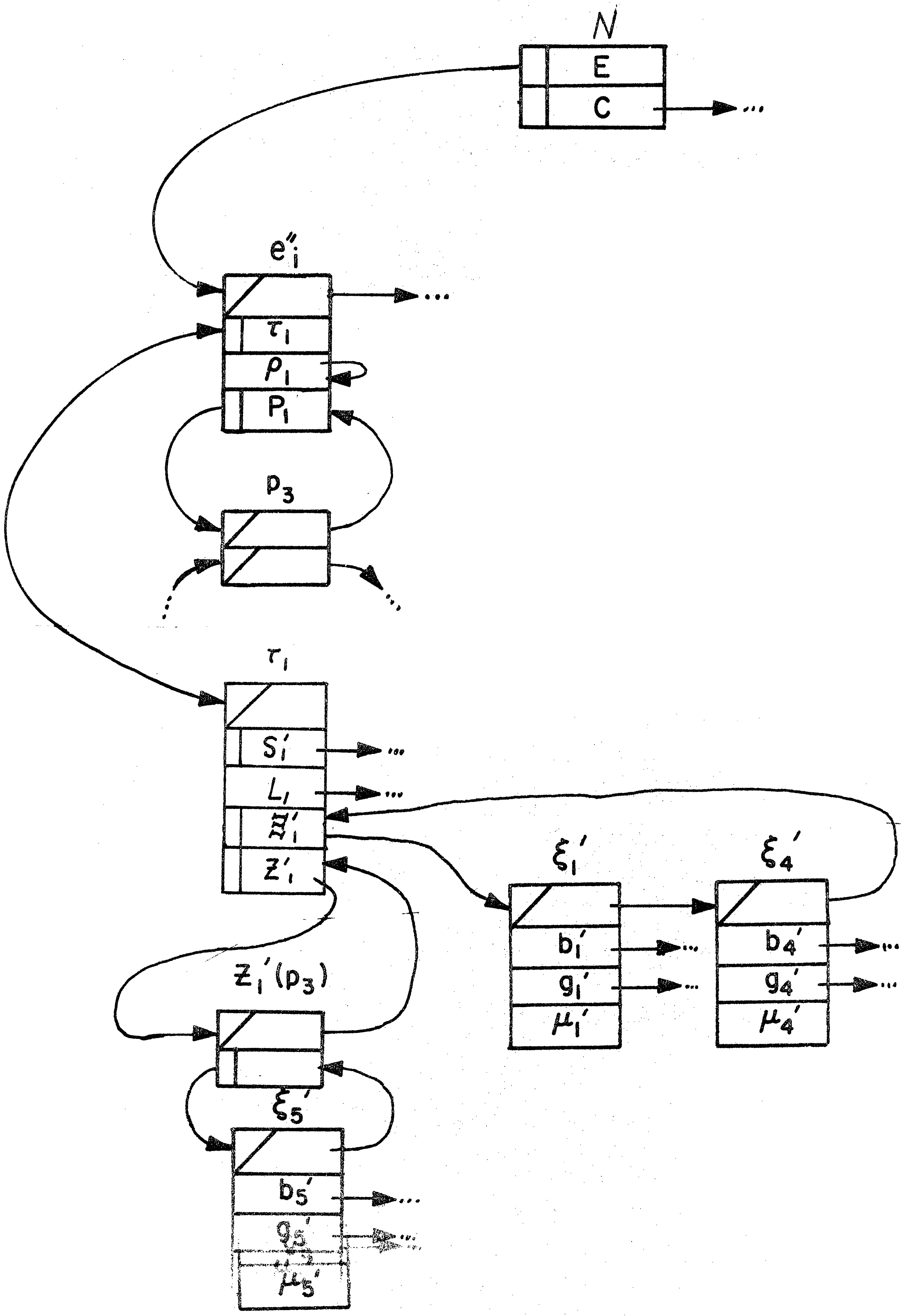
Generating Spontaneous Events

Figure 13



Additional Structure Generated by Spontaneous Event Routines

Figure 14



Fragment of Working Network Structure at Completion of Connection Absorption

Figure 15

Condition	Error Code
Illegal connection type.	X'00110000'
Connection joins ports of more than one element.	X'00230000'
Invalid type structure.	X'00240000'

SPON

SPON has one entry for each connection type. Each entry creates the spontaneous event set for the given element using the information contained in the connection parameter list and the ring of exoevents which has replaced the port ring. The correspondence between connection types and the SPON entry points is given below:

Connection Type	Type Number	SPON Entry Point
Simple Connection	X'81'	SPON81
Priority Branch	X'82'	SPON82
Random Branch	X'83'	SPON83

Calling sequence:

GPR 15:	Address of SPON entry point.
GPR 1:	Pointer to connection block.
GPR 12:	Base of working network area.

Upon return GPR 1 contains a pointer to the spontaneous event set. The QAS error routine DAMMIT is called with the indicated error code if any of the following situations arises:

Condition	Error Code
Coding for connection type not included in this version of SPON (i.e., an entry for this type exists in SPON but is not yet coded). Currently, there are no such entries.	X'00220000'
Parameter value missing	X'00250000'

In addition SPON83 checks to insure that the sum of the parameter values given for the random branch is equal to 1.0 within some limit of error. Currently, this limit is 0.01 and is defined by the constant called EPSILON. Consequently, if the sum of the parameter values is not equal to 1.0 $\pm$ EPSILON, DAMMIT is called with error code X'00250000'.

C. Trim State Set

The points in the rectangular "state set" S_R of the Markov chain of the entire network fall basically into three categories

(1) Forbidden states

(2) Recurrent states

(3) Transient states.

The forbidden states are those which immediately result in the occurrence of spontaneous events, which, in turn, take the process to a state which is not forbidden. The forbidden states are automatically pruned from the state set upon each consolidation of events and, hence, present no problem.

The difficulty arises with the presence of transient states within the model. Transient states are legitimate states of the Markov model, but since they are characterized by an equilibrium probability of zero and since they ordinarily have no bearing upon the calculation of equilibrium probabilities of other states, they are superfluous and waste valuable storage in the various QAS data structures.

Ideally, it would be desirable to identify transient states of consolidated elements as they are created, and to immediately remove them from the state set. Unfortunately, there appears to be no systematic procedure for identifying all of the transient states without performing a calculation very similar to the calculation of all equilibrium probabilities. This, of course, defeats the purpose of trying to remove them from the state set since it is desired to eliminate them and to reduce the state set <u>before</u> calculating the equilibrium probabilities.

Fortunately, there is a (relatively large) class of transient states which is relatively easy to recognize. It is these states which are eliminated by the "trim state set" operation.

The procedure for eliminating this class of transient states is the following: Subsequent to the absorption of each connection, those stat es which are not "target states" of any event are successively trimmed from the state set until no more such states can be found.

<u>TRIM</u>

The purpose of the routine TRIM is to eliminate that class of easily recognizable transient states from the state set of the element

(This includes suitably trimming the state sets of the various events, also.)

Calling sequence:

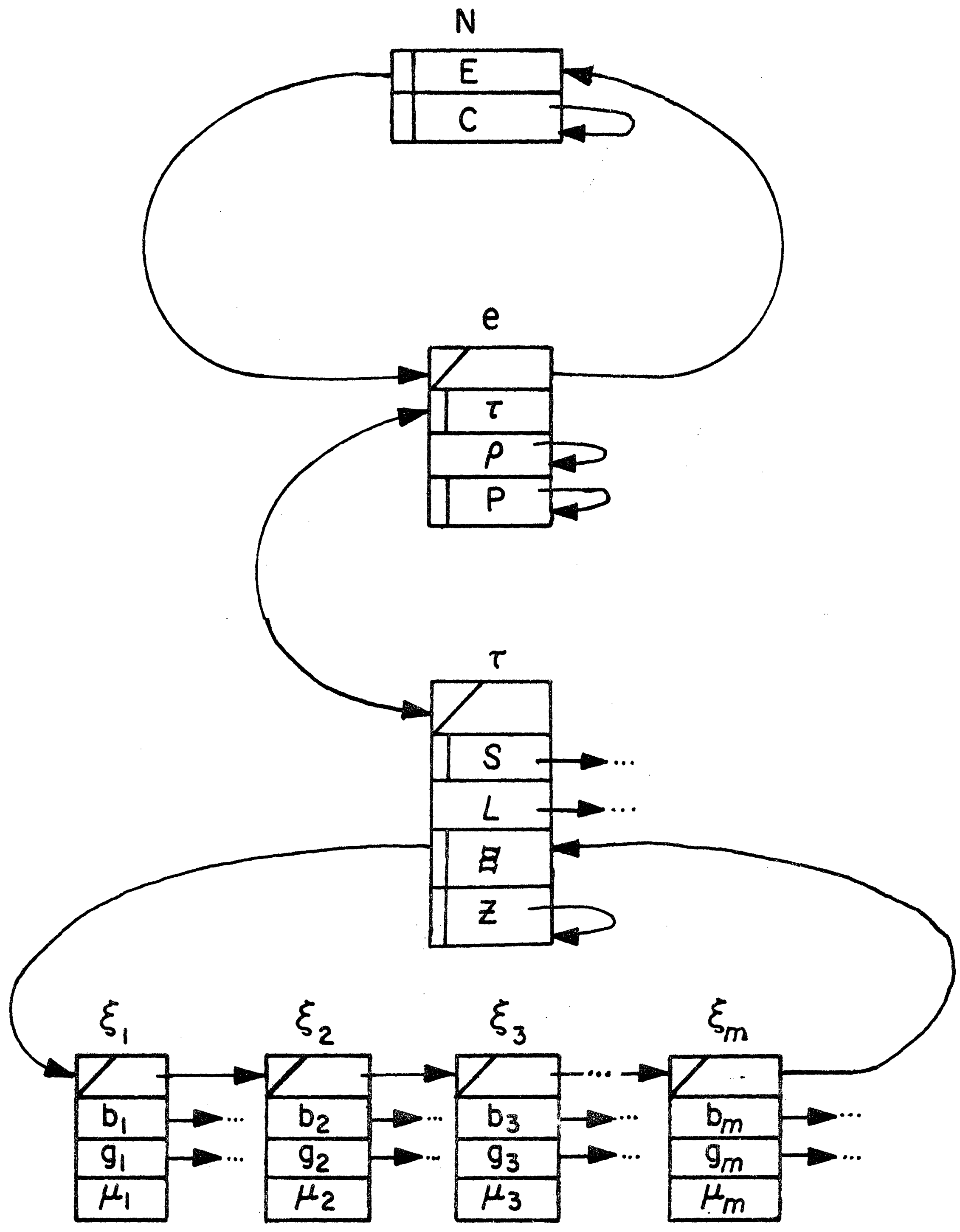
GPR 15:	Address of TRIM
GPR 1:	Pointer to element block (that element into which the connection has just been absorbed).
GPR 12:	Base of working network area.

D. Results Area

As mentioned earlier, the result of the compilation process is a network containing exactly one element having no ports or exo-events and having a large set of autoevents. The information in this set of autoevents describes the information in the matrix of transition intensities for the network. An example of how a typical compiled network might appear is shown in Figure 16. At this point, after making certain that one and only one element remains in the E-ring, subroutine GETRA is called to acquire the results area. Routines GETRA and PUTPRVEC, descriptions of which follow, are used both to acquire the results area and to create various data structures within it.

GETRA

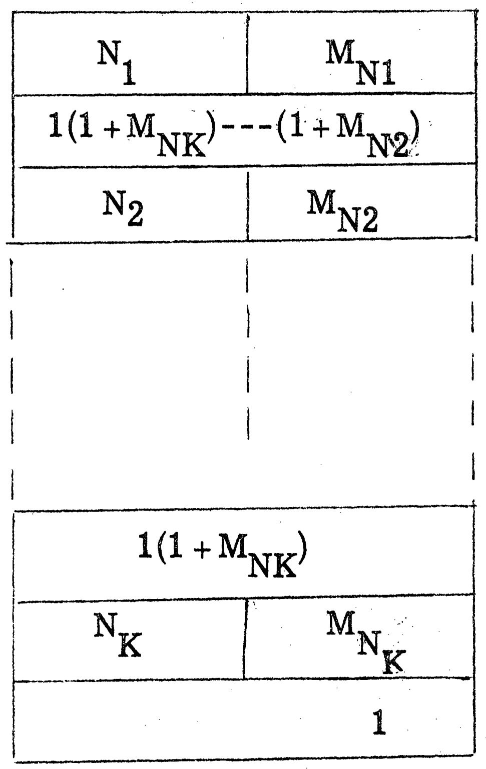
GETRA first scans through the state set S of the remaining element (Figure 16) to find the largest value on each dimension of the state set of the original network. It then forms the partial products of these values and creates the mapping structure shown below:



Typical Result of Absorption of All Connections

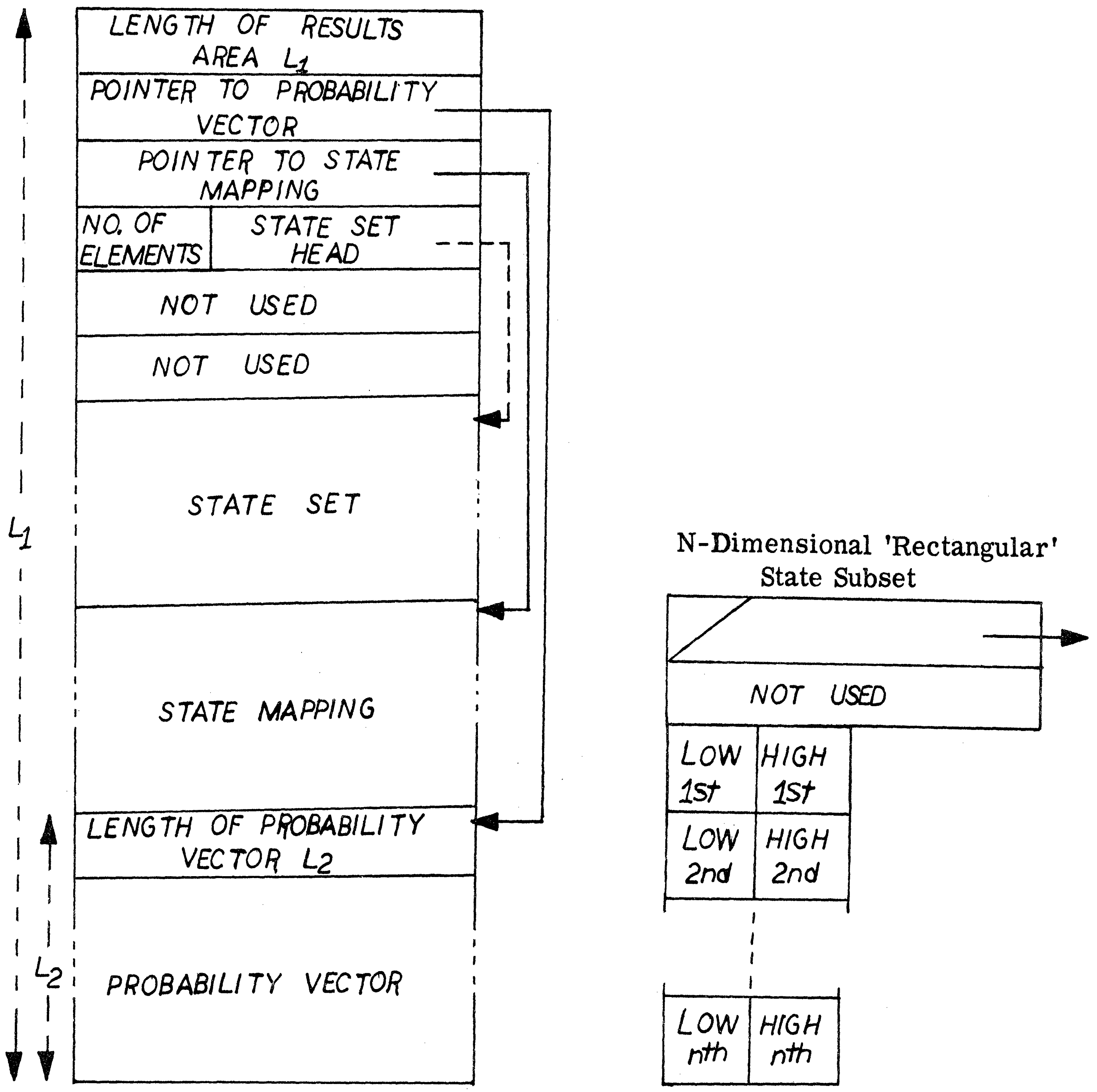
Figure 16

Mapping



M_{N_K} : Maximum value in the state set corresponding to the element named N_K.

Next it computes the amount of storage necessary to represent the state set S of the remaining element, the mapping structure, and the steady-state probability vector, and acquires it. Finally it creates the structure shown in Figure 17. Note that the format of each 'rectangle' in the state set is slightly modified by the addition of a full word after the pointer. This word is not used currently, but is reserved to contain the base index of each rectangle in case the future modification deem this to be necessary. At this point (in Figure 17) only the state set and the mapping structure have been inserted into the results area, and the block for probability vector is empty.



Results Area

Figure 17

Actually this block of storage, along with working probability area, is used by the SOLVE routine during the iteration process, and after the probability vector has been computed, it is inserted into the storage reserved for it in the results area.

Calling sequence:

GPR 15:	Address of GETRA
GPR 1:	Pointer to type block
GPR 12:	Base of working network area

PUTPRVEC

This routine inserts the probability vector into the storage reserved for it in the results area.

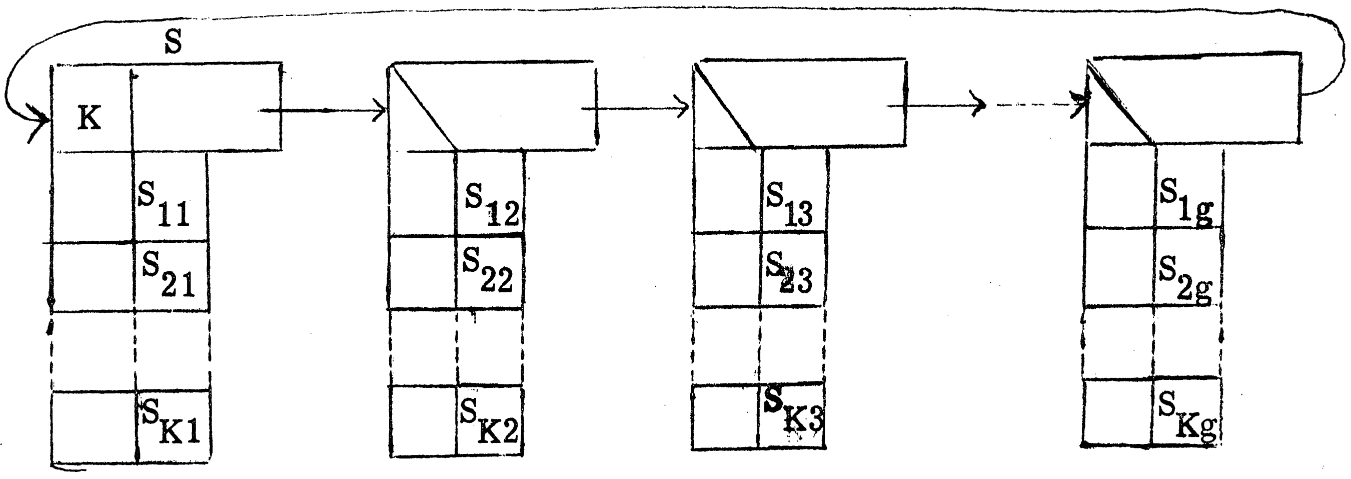
Calling sequence:

GPR 15:	Address of PUTPRVEC
GPR 1:	Address of probability vector to be inserted

E. Calculation of Equilibrium Probabilities

The calculation of equilibrium probabilities is initiated, after the network has been compiled, by a call from COMPSOLV to a routine call SOLVE. SOLVE using an iterative procedure very similar to the one used RQA-1 [3] takes the single element resulting from the compilation and produces the steady state probability vector for the original model. Initially, two vectors - one in the results area and the second in the working probability area - are obtained by COMPSOLV and passsed on to SOLVE. The first word in each vector is its length, including the length word. The rest of words correspond to states of the element and contain that state's probability. The mapping used to compute the linear index of a state in the multi-dimensional state set of the element (and, hence, the state's position in each of the two vectors) is given below:

Let S be the state set and let,



$$S_1 = \text{Max}\ \{S_{11}, S_{12}, \ldots, S_{1g}\} + 1$$

$$S_2 = \text{Max}\ \{S_{21}, S_{22}, \ldots, S_{2g}\} + 1$$

$$S_K = \text{Max}\ \{S_{K1}, S_{K2}, \ldots, S_{Kg}\} + 1$$

Then the linear index I_B for a particular state B is:

$$I_B = b_k + b_{K-1} S_K + b_{K-2} S_{K-1} S_K + \\ + b_1 S_2 S_3, \ldots, S_K$$

where B is given by the tuple

$$B = \begin{array}{|c|} \hline b_1 \\ \hline b_2 \\ \hline \vdots \\ \hline b_K \\ \hline \end{array}$$

Before the iteration process to determine the steady state probabilities begins, the probability intensities, μ, for each autoevent of the element are normalized as follows: One of the probability vectors is cleared to zero. Then the probability intensity, μ, for each autoevent is added to each of the initial-state entries for that autoevent. The largest entry in the vector is found and all probability intensities are normalized to slightly under this as follows:

$$\mu_i = \mu_i \times .99/\text{largest entry}$$

Then each probability vector is cleared (i.e., each entry is set to zero). Next 1.0 is put into each entry that corresponds to the initial-state (b) or final-state (b+g) of any autoevent. The vectors are then normalized so that the sum of the probabilities of all states is 1.

Then the iteration process is performed. At the beginning of each iteration the probability vectors have identical entries, normalized to a sum of 1.0. One is designated P_{old} and one P_{new}. Then for each autoevent the following operations are performed.

$$P_{new}(b) = P_{new}(b) - 1\ P_{old}(b)$$

$$P_{new}(b+g) = P_{new}(b+g) + \mu\ P_{old}(b)$$

where: the index b represents an initial-state of the autoevent, the index b+g represents the final-state corresponding to the initial-state b, and μ is the normalized probability intensity of that autoevent.

One iteration then consists of the above operation for each initial-state and each final-state of each autoevent.

At the end of each iteration P_{new} is normalized to 1.0 The iteration process is discontinued if:

Iteration limit exceeded --	A comment is then sent to SPRINT to that effect. SOLVE returns with GPR 1 pointing to P_{new}.
Convergence achieved --	That is, the maximum difference between any corresponding entries in P_{old} and P_{new} is less than epsilon (FPR 0). SOLVE returns with GPR 1 pointing to P_{new}.

Otherwise, another iteration is made.

Calling sequence:

GPR 15:	Address of SOLVE
GPR 0:	Maximum number of iterations
GPR 1:	Pointer to element block
GPR 2:	Address of probability vector in results area
GPR 3:	Address of working probability vector
GPR 12:	Address of working network area
FPR 0:	Epsilon, convergence criterion

On return GPR 1 points to the probability vector.

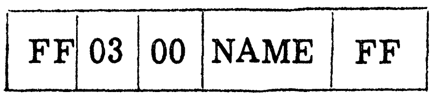
V. Result Phase Routines

Result phase consists of a collection of routines which operate on the results area to compute and format requested results for display as graphs or printed tables. Even though there is sufficient information in the results area to compute expected values and marginal probability functions for the states of various elements, at this point in time only the probability density function for the entire model, or for a single element in the model, can be computed and displayed. The result phase routines include the result phase interpreter, routines to plot and label graphs, and finally routines to print the results in tabular form.

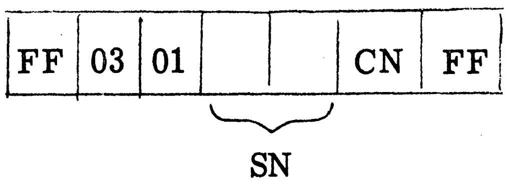
A. Result Phase Interpreter

The result phase interpreter is a routine named RESULTS, which is called by the QAS supervisor whenever a result phase command is received, that is, whenever the first byte of the command (excluding the delimiting bytes) is X'03'. As noted earlier, the supervisor insures that the current model, the results for which have been requested, has already been compiled and solved. RESULTS checks to make sure that the results area, indeed, exists and loads GPR 12 with its address. The result phase commands, as interpreted by the result phase, have the following formats and interpretations. Each division in a command corresponds to one byte and all numbers contained therein are hexadecimal:

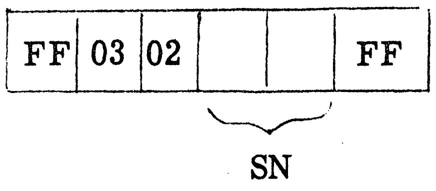
Plot Probability Density Function



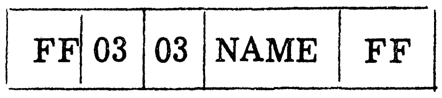
Modify Current Plot



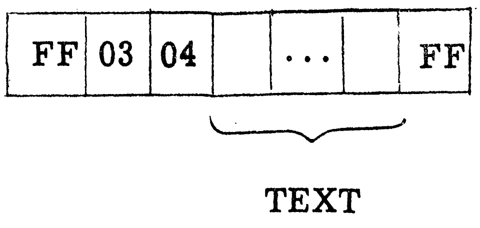
Display Coordinates



Type Probability Density Function



Call User-Supplied Routine



Note that the last command is not currently generated by SELMA. The abbreviations used within these commands have the following meanings:

NAME	-	A number between 1 (X'01') and 127 (X'7F') identifying a particular element. Element name X'00' specifies the entire model.
CN	-	A number which specifies the number of points to be plotted.
SN	-	A two-byte number between 0 and 16,363 which specifies the state of the element or the linear index of state of the entire model. This number is encoded so that the low-order bit in each byte is zero and is obtained by concatenating the high-order 7 bits of the two bytes.

Both the modify and display commands refer to the previous plot and no additional computation of results is necessary. For these cases the parameters in the commands are simply decoded and the plotting routines EXPAND and DISPLAY, respectively, are called. However, for the other two commands (excluding the call-user-supplied-routine command), unless the probability density function for the entire model is requested, additional computation is done by RESULTS to generate a vector of probabilities corresponding to the states of the indicated single element. For the entire model, such a vector already exists in the results area. The last step taken by RESULTS is to call either the plotting or the printing routine, as indicated by the command.

Calling sequence:

GPR 15:	Address of RESULTS
GPR 1:	Address of one word parameter list.

Parameter List:

Word 1:	Location of first byte of command

B. Plotting Routines

The plotting routines are used to display results via SELMA as histograms, and to provide support for other facilities provided by SELMA, such as obtaining numerical values for points on the graph and looking at various sections of the graph in detail, all of which is explained in detail in the companion report on SELMA [2]. A maximum of 50 points may be plotted at any one time. Result plots which exceed this limit may be displayed by making use of the modify command. The plotting system consists of the three subroutines PLOT, EXPAND, and DISPLAY, descriptions of which follow below.

PLOT

This routine is used both to initialize the plotting system and to generate a plot. PLOT saves the address and the length of the vector of probabilities to be plotted for later calls to EXPAND and generates a plot by a call to EXPAND with a starting index of zero and the default value of 50 as a count.

Calling sequence:

GPR 15:	Address of PLOT
GPR 1:	Location of 4-word parameter list

Parameter List:

Word 1:	Address of vector to be plotted
Word 2:	Location of a full word containing number of elements in vector to be plotted
Word 3:	Location of a full word containing number of characters in label for y-axis
Word 4:	Location of first byte of label for y-axis. (x-axis label is supplied by SELMA.)

PLOT is a FORTRAN callable routine and can be called as follows.

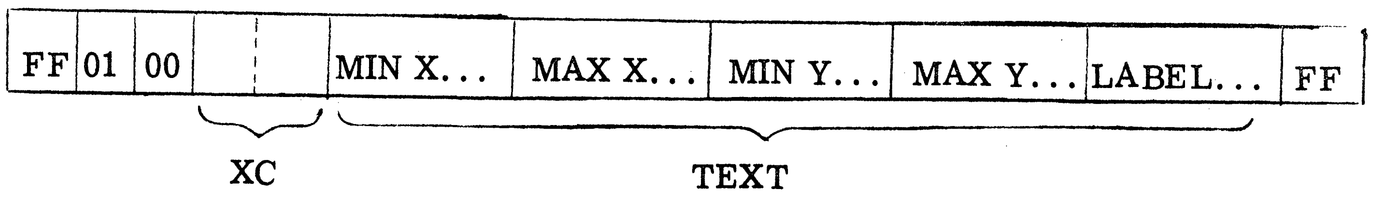
```
CALL PLOT (VEC, VECSZ, LABSZ, 'LABEL')
```

where VEC is the vector to be plotted, VECSZ is the size, and LABSZ is the length of the label.

EXPAND

EXPAND may be used to plot any portion of the vector address and size of which were specified by the last call to PLOT. The particular portion to be plotted is specified by a starting index and a count of the number of points therein. If the count is zero or greater than fifty, fifty points are plotted. The limits of the vector are, of course, never exceeded (i.e., if the sum of the count and the starting index indicates some limit beyond the end of the vector, then only the points which are in the vector are plotted). Given a starting index, a count, and the address of the vector, EXPAND first computes the starting address of the section of the vector to be plotted. Next a search for the largest value in this section is made. If this value is greater than 1.0, then it is used to normalize the rest of the points in the section; otherwise the least negative power of 2 greater than the maximum value is used. Finally, the minimum and maximum values of the abscissa are computed from the starting index and count. On the basis of this information, the following two commands for SELMA are generated and inserted into the output buffer by a call to QASWRITE.

Set Up Graph



where

XC is the abscissa increment encoded the same way as SN. (2 bytes)

TEXT consists of five variable length fields specifying the minimum X value, the maximum X value, the minimum Y value, the maximum Y value, and the Y-axis label. Each of these fields consists of a sequence of bytes, the first of which contains the number of bytes which follow it in the field and the remainder of which are SEL 6-bit character codes specifying the appropriate item.

Plot values

FF	01	01	Y	C	Y	C	...	Y	C	FF

where

YC is a coded normalized ordinate value (2 bytes).

Calling sequence:

GPR 15: Address of EXPAND

GPR 1: Starting index

GPR 0: Count.

<u>DISPLAY</u>

DISPLAY is used to determine and display the coordinates of a point on a graph. The index of the point to be displayed is supplied as a parameter. DISPLAY inserts the following record into the output buffer.

Display Coordinates

FF	01	02	X VALUE	Y VALUE	FF

$\underbrace{\text{X VALUE} \quad \text{Y VALUE}}_{\text{TEXT}}$

where X VALUE and Y VALUE are variable length fields like the TEXT fields for the command "Set Up Graph" containing SEL 6-bit codes specifying the coordinates.

Calling sequence:

GPR 15: Address of DISPLAY

GPR 1: Index of point the coordinates of which are to be displayed.

C. Printing Routines

The printing routines are used to print (via teletype, for instance) the probability density function for the entire model or for a single element. When the probability function for the entire model is requested, both the linear index and the n-tuple of state is printed. These routines have been written in FORTRAN and they put the output records on logical I/O unit 6.

PRNTRS (SIZE, VEC, NAME)

PRNTRS is used to print the probability density function for a single element named NAME. The states of the element are given by the set $\{0,1,2,3,\ldots,\text{SIZE-1}\}$ and the corresponding probabilities are contained in the vector VEC.

PRNTAL (VEC, SIZE, NAMVEC, DIMVEC)

PRNTAL is used to print the steady state probability vector for the entire state space of the model. VEC and SIZE define this probability vector, while NAMVEC and DIMVEC define the list of element names in the model and their corresponding maximum dimensions, and are used to compute the n-tuple of state corresponding to each probability.

VI. Documentation Phase Routines

The purpose of the documentation phase routines is to save/retrieve the SELMA network display structure and the corresponding QAS network structure (network area) on/from a user's supplied file. Optionally, for a compiled and solved model, the results area may also be saved. Rather than saving the entire display structure, an encoded version of it retaining just enough data to enable the model to be regenerated is saved. The information which is saved is produced by SELMA. QAS does nothing except save the records containing this information in a file, and later on, after the retrieval command, these records are transferred back to SELMA. The details of these records and how they are manipulated to redraw the model are explained in the companion report on SELMA [2]. The reason for not saving the entire display structure is that all the traffic between SELMA and QAS is via a relatively slow 201 line (2000 bits/sec.). As a result, saving the entire display structure would be too time-consuming. QAS, of course, has the responsibility of saving and retrieving the network area and the results area (if necessary). QAS documentation phase consists of the main control section DOCUMENT and the internal service routines STSAVE, SAVERC, ENDSAVE, and GET.

DOCUMENT

DOCUMENT examines the second byte of the documentation phase command (excluding the delimiter) and calls the appropriate service routine as indicated in Table 13. In Table 13 FILENAME corresponds

Table 13

Documentation Phase Commands

Command	Service Routine
$\boxed{\text{FF}}\boxed{\text{04}}\boxed{\text{00}}\underbrace{\boxed{\text{FILENAME}}}_{\text{SEL 6-bit Code}}\boxed{\text{FF}}$	STSAVE
$\boxed{\text{FF}}\boxed{\text{04}}\boxed{\text{01}}\boxed{\text{FF}}$	ENDSAVE
$\boxed{\text{FF}}\boxed{\text{04}}\boxed{\text{02}}\underbrace{\boxed{\text{FILENAME}}}_{\text{SEL 6-bit Code}}\boxed{\text{FF}}$	GET
$\boxed{\text{FF}}\boxed{\text{04}}\boxed{\text{03}}\underbrace{\boxed{\text{BINARY TEXT}}}_{\text{SELMA phase 02 command}}\boxed{\text{FF}}$	SAVERC

to the name of the user's supplied file, and the BINARY TEXT corresponds to the encoded display structure data, which will be transmitted back to SELMA during the retrieval of this model and from which SELMA will redraw the network.

Calling sequence:

GPR 15:	Address of DOCUMENT
GPR 1:	Address of one-word parameter list

Parameter list:

Word 1:	Location of documentation phase command

<u>STSAVE</u>

Since the file name is transmitted to QAS in SEL 6-bit code, it has to be translated to its EBCDIC equivalent (an MTS [4] requirement). A modifier is associated with the file name and it specifies whether the results area is to be saved or not. If the file name is followed immediately by the delimiting character X'FF', the modifier is assumed to be off, which implies that the results area is not to be saved. If the file name is followed by a blank (this can be detected since a file name cannot contain a blank character) and then the characters 'ALL', the modifier is considered to be on, and implies that the results area is to be saved. Next STSAVE checks to make sure that the specified file exists and if it does exist, STSAVE opens it. If the file does not exist, patchup mode is entered and the name of another file (which hopefully does exist) is requested. After the file has been opened, it is emptied and the pointer to the file control block is saved for future accesses to the file. At this point the following command is sent back to SELMA (through QASWRITE) to indicate that the file has been opened and that SELMA should start transmitting network display records.

Transmit Network Display Records

FF	03	00	FF

SAVERC

The first three bytes of the command received from SELMA are stripped and replaced with the delimiter X'FF'. This is then saved in the already opened file. Thus the initial records in the documentation file are SELMA phase 02-commands and the first two bytes of all these records are X'FF02'. (This information is used during the retrieval process.) This process of saving display records is continued until the end of display command is received from SELMA, at which point the service routine ENDSAVE acquires control.

ENDSAVE

ENDSAVE computes the number of records contained in the network area and, if necessary, the results area. Each record (corresponding to a line) is assumed to be equal to 240 bytes. Before inserting the network and results area into the file, the following record is written.

Bookkeeping Record:

Code	NASZ	RASZ

where each of the three fields is 4 bytes long and

Code: contains X'FFFFFFFF'. Thus, during retrieval the second byte of each command can be examined to check for the end of the display records. (Normally, this byte cannot be X'FF'.)

NASZ: contains the number of lines necessary to save network area.

RASZ: contains the number of lines necessary to save results area. If the results area is not to be saved, this word is set to zero.

Then the records corresponding to the network area and the results area are written and the file is closed.

<u>GET</u>

GET is the routine used to service the retrieval command from SELMA. GET, like STSAVE, translates the file name and opens it. If the file does not exist, an appropriate patchup request is generated. As noted earlier, the initial lines in the file are display records, which are subsequently followed by the bookkeeping record. Thus, the lines in the documentation file are read sequentially and inserted into the output buffer (QASWRITE) until the bookkeeping record (which can be recognized by X'FF' in the second byte) is encountered. Noted that the display records are not transmitted directly to SELMA but, rather, are inserted into the the output buffer to be retrieved by SELMA at its convenience. From the information contained in the bookkeeping record the storage necessary to reclaim the network area and the results area is computed. The previous network area (if any) is released and a new network area is acquired into which the network area records are read. The previous results area (if any) is destroyed regardless of whether a new one is to be acquired or not. If the results area was originally saved in the file, it is retrieved.

VII. Conclusions

The implementation of QAS and SELMA [2] has demonstrated the feasibility and the usefulness of a programming system for the conversational design of stochastic service systems using a graphical display for both specifying the stochastic network and evaluating it. The networks, which are restricted to systems which can be modeled by a continuous time, finite Markov chain, are solved by numerical solution of the Kolmogorov equilibrium equations. The advantages of this approach, in terms of speed, precision, and ease of design, have been demonstrated. However, before this system can be used as a handy tool by the queueing analyst, a few modifications to the present system will be necessary both to improve its performance and to enhance its capability.

As noted earlier, considerable thought was given during the implementation of this system to making the definitions of new elements or connections a relatively easy task. The set of primitive elements and connections which is currently available (i.e., queue, server, exit, source, and simple, priority and random connections) is not extensive enough to allow the specification of certain important models. Hence, the definition of any additional elements or connection types would be very helpful in extending the capabilities of the system. The routines which would require modification in order to include any new definitions are CREAT, ASSPAR, and ALTER. Additional type structure routines and spontaneous event routines would also be necessary. The inclusion of new elements in SELMA would not be difficult, since the "menu" of elements in SELMA is table driven. Another desirable feature would be allowing the user to define composite elements using the existing primitive elements. Such a facility could conceivably be supported exclusively by SELMA.

The current system is rather limited in the kinds of results that can be displayed. Subroutines to compute expected values, marginal probability functions, etc. could easily be implemented. However, a universally acceptable set of results satisfying the majority of users is difficult, if not impossible, to define. Perhaps a facility whereby a user supplied routine is used to compute results would be the most viable alternative. An interesting possibility would be a post-processing system which, using the QAS documentation file, would generate via the Calcomp plotting system [4] a hard copy of the network diagram and plots of useful results.

The weakest features in the current system are the lack of efficient state mapping and deferred evaluation schemes. In general the cartesian product of the states of all elements in the model can be partitioned into two sets: the set of actual states of the model and the set of ordered tuples which are not states. The simple mapping used in SOLVE routine maps the entire cartesian product of the states of all elements to the set of consecutive integers $\{0,1,2,\ldots,N\}$ thereby making the size of the probability vector unnecessarily large. Moreover, the matrix of transition intensities descriptive of the model is never explicitly derived. Instead the autoevent set is used during the iteration process, necessitating repeated calculations of the linear index, thereby rendering SOLVE less efficient. A very desirable feature in QAS would be a system of deferred evaluation whereby, after compilation, each intensity in the autoevent set is given by an algebraic expression in terms of the parameters of the model, which expression is evaluated when the value of the given intensity is required for some calculation and after the parameter values are supplied. Thus a model with undefined parameters could be compiled and the results of compilation could be used for a number of different parameter values. Currently changing a parameter value necessitates a recompilation.

References

1. Wallace, V. L., and J. B. Irani, A System for the Solution of Simple Stochastic Networks, Technical Report 14, Concomp Project; also SEL Technical Report 31, Systems Engineering Laboratory, The University of Michigan, Ann Arbor, September 1969.

2. Jackson, J. H., SELMA: A Conversational System for the Graphical Specification of Markovian Queueing Networks, Technical Report 23, Concomp Project; also SEL Technical Report 45, Systems Engineering Laboratory, The University of Michigan, Ann Arbor, October 1969.

3. Wallace, V. L., and R. S. Rosenberg, RQA-1, The Recursive Queue Analyzer, SEL Technical Report 2, System Engineering Laboratory, The University of Michigan, Ann Arbor, February 1966.

4. University of Michigan Computer Center, MTS: Michigan Terminal System, Ann Arbor: University Press, December 1967.

5. Wallace, V. L., and K. B. Irani, Network Models for the Conversational Design of Stochastic Service Systems, Technical Report 13, Concomp Project; also SEL Technical Report 30, Systems Engineering Laboratory, The University of Michigan, Ann Arbor, November 1968.

Appendix A

Format of Commands

This appendix lists all the commands which may be transmitted between QAS and SELMA. The formats of the commands which are accepted by QAS are indicated by Table 14, and the formats of the commands which are accepted by SELMA are indicated by Table 15. The abbreviations for various groups of data bytes are interpreted as follows:

CN	Connection name. A number between 129 and 254 which identifies a particular connection. (1 byte)
CPN	Connection port number. A number between 1 and 254 which specifies a particular port of the connection specified by an associated connection name. (1 byte)
CT	Connection type. A number between 129 and 254 which specifies the type of connection (e.g., simple connection, random branch or merge, priority branch or merge). (1 byte)
DW	Display file word. Two bytes, each of which has a value from 0 through 127. These two bytes are decoded to form an 18-bit word to be loaded into core by SELMA according to the following scheme: (1) The low-order 7 bits of each byte are concatenated to form a 14-bit number.

(2) The two high order bits of the 14-bit number are placed into positions 0 and 1 of the 18-bit word, and the remaining 12 bits are placed into positions 6-17 of the 18-bit word. Positions 2-5 of the 18-bit word are set to zero.

EN	Element name. A number between 1 and 127 which identifies a particular element. (1 byte)
EPN	Element port number. A number between 1 and 254 which specifies a particular port of the element specified by an associated element name. (1 byte)
ET	Element type. A number between 1 and 127 which specifies the type of element (e.g., queue, server, or source or exit). (1 byte)
FN	File name. A sequence of bytes whose values are SEL 6-bit character codes which represent a file name (1 to 16 bytes)
GPN	Generation parameter number. A number between 1 and 254 which identifies a particular generation parameter (i.e., parameter which modifies an element or connection type). (1 byte)
GPV	Generation parameter value. A number between 1 and 254 which represents the value of a generation parameter. (1 byte)

IL	Iteration limit. Two bytes, each of which has a value between 0 and 127. A number between 1 and 16,383 which represents the maximum number of iterations which will be performed whenever a model is solved is obtained by concatenating the low order 7 bits of the two bytes.
LET	Local element type. A number between 1 and 254 which identifies a graphical symbol for an element type. This number is not necessarily the same as the corresponding element type, for several graphical symbols may be associated with one element type (e.g., source and exit represent the same element type). (1 byte)
PN	Parameter number. A number between 1 and 254 which identifies a parameter for an element or connection. (1 byte)
PV	Parameter value. A sequence of bytes whose values are SEL 6-bit character codes which represent a non-negative real or integer number. (1 to 18 bytes)
SC	State count. A number between 1 and 254 which represents the number of points to be plotted. If this number is either zero or greater than 50, 50 points are plotted (1 byte)

SN	State number. A number between 0 and 16, 383 which represents an abscissa on a graph plotted by QAS. This number is coded in the same way that an iteration limit (IL) is coded. (2 bytes)
T	Text item. A sequence of bytes, the first of which has a value which is the number of bytes which follow it, and the remainder of which are SEL 6-bit character codes. (2-16 bytes)
WC	Word count. A number between 1 and 16, 383 which represents the size of storage block required to load a connection leaf when retrieving a model from a file. This number is coded in the same way that an iteration limit (IL) is coded. (2 bytes)
XC	An abscissa between -8192 and 8191. This coordinate is coded in the same way that an iteration limit (IL) is coded. However, the 14-bit number represented is interpreted as a two's complement number, rather than as an unsigned positive number. (2 bytes)
YC	An ordinate between -8192 and 8191. This number is coded in the same way an abscissa is coded. (2 bytes)

Phase	Command		
Byte	Byte	Data Bytes	Function
00	00	---	Null
00	01	---	Patch-up[1]
00	02	---	Call system
00	03	---	Call error[1]
00	04	---	Call MTS[1]
00	05	---	Initialize
00	06	---	Wipe out QAS output buf
01	00	EN or CN, ET or CT, GPV	Create element
01	01	EN or CN	Destroy element or connection
01	02	EN or CN, PN, PV	Assign parameter valu
01	03	CN, CPN, EN, EPN	Connect
01	04	EN, EPN	Disconnect
01	05	EN or CN, GPN, GPV	Alter generation parameter value
02	00	IL, PV[2]	Compile and solve[1]
03	00	EN[3]	Plot results
03	01	SN, SC	Modify plot
03	02	SN	Get value for graph
03	03	EN[3]	Type results
04	00	FN	Begin saving model
04	01	---	Terminate saving mode
04	02	FN	Retrieve model from file
04	03	SELMA phase 02 command	Save command to be returned

Table 14. Commands accepted by QAS.

[1] This command is not currently generated by SELMA.

[2] Parameter value specifies convergence factor.

[3] An element name 00 specifies the entire model.

Phase Byte	Command Byte	Data Bytes	Function
00	00	---	Null
00	01	(See text)	Patch-up
00	02	---	End of file
01	00	EX,T,T,T,T,T[1]	Set up graph
01	01	YC,YC,...,YC	Plot values
01	02	T	Display single value
02	00	YC,XC	Create fragment[3]
02	01	YC,XC,EN,00,LET	Create element[3]
02	01	YC,XC,EN,00,WC	Create connection[3]
02	02	DW,DW,...,DW	Load connection segment[3]
02	03	---	Insert connection leaf[3]
02	04	YC,XC,PV[3]	Assign parameter [3]
02	05	EN,EPN	Connect[3]
02	00	---	Send model to be saved.

Table 15. Commands accepted by SELMA.

---

[1] Distance between abscissas, minimum x label, maximum x label, minimum y label, maximum y label, y axis label.

[2] Ordinate label

[3] See SELMA report [2].

Appendix B

Miscellaneous MTS Routines

For the sake of completeness, the MTS library routines called explicitly by QAS will be listed here with a brief mention of their functions. This list does not contain some of the FORTRAN supplied routines.

GETSPACE

Obtains storage under program control.

FREESPAC

Releases storage that was obtained with GETSPACE.

SYSTEM

Produces a call to monitor system MTS and releases all storage occupied by the calling program.

ERROR

Returns control to MTS to terminate execution. The comment "ERROR RETURN" is printed. The storage occupied by the calling program is not released.

MTS

Returns control to MTS. Control can be transferred back to calling program by the MTS command $RESTART.

CANREPLY

Finds out if user is at a terminal or if this is a batch job.

GETFD

Obtains a file or a device. Returns pointer to file/device control block, which is used in subsequent calls to file management routines.

GDINFO

Obtains information about a file or a device and finds out if a file exists or not.

FREEFD

Frees a file or device obtained with GETFD.

EMPTY

Removes all lines from a file without destroying it.

REWIND#

Resets a magnetic tape or a file without destroying it.

SDUMP

Produces a hexadecimal dump of any or all of the following: 1. general registers, 2. floating point registers, and 3. a specified region of core storage.

SCARDS

Reads an input record from logical unit SCARDS.

GUSER

Reads an input record from logical unit GUSER, the default value of which is master source (SELMA).

SPRINT

Writes an output record on logical unit SPRINT (sink).

SERCOM

Writes an output record on the logical unit SERCOM (sink).

READ

Obtains an input record from a specified logical unit.

WRITE

Writes an output record on a specified logical unit.

unclassified

Security Classification

153

DOCUMENT CONTROL DATA - R & D

(Security classification of title, body of abstract and indexing annotation must be entered when the overall report is classified)

1. ORIGINATING ACTIVITY *(Corporate author)* The University of Michigan Concomp Project	2a. REPORT SECURITY CLASSIFICATION Unclassified
	2b. GROUP ---

3. REPORT TITLE

An Implementation of the Queue Analyzer System (QAS) on the IBM 360/67

4. DESCRIPTIVE NOTES *(Type of report and inclusive dates)*
Technical Report May 1970

5. AUTHOR(S) *(First name, middle initial, last name)*

L. S. Randall G. A. McClain
I. S. Uppal J. F. Blinn

6. REPORT DATE May 1970	7a. TOTAL NO. OF PAGES 152	7b. NO. OF REFS 5

8a. CONTRACT OR GRANT NO. DA-49-083 OSA-3050	9a. ORIGINATOR'S REPORT NUMBER(S) Technical Report 22
b. PROJECT NO.	
c. ---	9b. OTHER REPORT NO(S) *(Any other numbers that may be assigned this report)* Systems Engineering Laboratory Report 07842-4-T
d. ---	

10. DISTRIBUTION STATEMENT

Qualified requesters may obtain copies of this report from DDC

11. SUPPLEMENTARY NOTES	12. SPONSORING MILITARY ACTIVITY Advanced Research Projects Agency

13. ABSTRACT

This report details and documents QAS, a conversational programming system composed of an aggregation of programs and data structures resident in the IBM 360/67 which accepts graphical descriptions of Markovian queueing networks via data-phone from a remote graphical system resident in a DEC 339, and which returns solutions to these networks to the remote system according to requested specifications.

DD FORM 1 NOV 65 1473

Security Classification

14. KEY WORDS	LINK A		LINK B		LINK C	
	ROLE	WT	ROLE	WT	ROLE	WT
Computer-Aided Design						
Computer Graphics						
Computer Languages						
Queueing Networks						
Markovian Networks						
Network Models						
Numerical Queueing Theory						
Mathematical Modeling						
Data Structures						
Large Scale Systems						
Conversational Design						